NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

AN AUTONOMOUS

PLATFORM SIMULATOR (APS)

by

Larry R. Shannon and William A. Teter

June 1989

Thesis Advisor: Robert B. McGhee

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Report Security Classification: UNCLASSIFIED	1b Restrictive Markings
Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
Declassification/Downgrading Schedule	
Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (*If Applicable*): 52	7a Name of Monitoring Organization: Naval Postgraduate School
Address (*city, state, and ZIP code*): Monterey, CA 93943-5000	7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
Name of Funding/Sponsoring Organization; 8b Office Symbol (*If Applicable*)	9 Procurement Instrument Identification Number
Address (*city, state, and ZIP code*)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

Title (*Include Security Classification*)
AN AUTONOMOUS PLATFORM SIMULATOR (APS)

Personal Author(s)
Larry R. Shannon and William A. Teter

Type of Report	13b Time Covered	14 Date of Report (*year, month,day*)	15 Page Count
Master's Thesis	From To	June 1989	202

Supplementary Notation
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
ld	Group	Subgroup	Moving Platform Simulators, Visual Simulators, Real-Time Graphics, Distributed Processing, Line-of-Sight, Real-Time Path Planning

Abstract (*continue on reverse if necessary and identify by block number*)
The development of an intelligent autonomous vehicle, that can perform high risk missions or operate in vironments too hazardous for humans, has been a long standing quest of the military community. The tonomous Platform Simulator (APS) uses the flexibility and power of realistic graphical simulation to provide a v cost testbed for the study of real-time path planning algorithms and control strategies without the commitment resources involved in building a prototype system. It is a bridge between the theoretical study of an abstract AI th planning problem and applied research, producing concrete performance measurements under realistic nditions.

APS consists of one or more vehicle simulators, each implemented on a Silicon Graphics IRIS/4D-70GT aphics workstation. One vehicle simulator is linked with an AI agent path planner, implemented on a pair oa mbolics AI workstations using the Automated Reasoning Tool development shell.

System trails demonstrated that APS was able to achive real-time path planning and guidance of a realistically picted ground vehicle navigating using digitized data of actual terrain. Communication bottlenecks currently hit the ability to make direct comparisons between human and machine control, but the system holds promise to the gap as a pre-prototype autonomous platform simulator.

Distribution/Availability of Abstract	21 Abstract Security Classification	
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	UNCLASSIFIED	
Name of Responsible Individual: Prof. Rober B. McGhee	22b Telephone (*Include Area code*): (408) 646-2449	22c Office Symbol: Code 52Mz

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

AN AUTONOMOUS PLATFORM SIMULATOR (APS)

by

Larry Richard Shannon
Captain, United States Marine Corps
B.S., University of Washington, 1981

and

William Albert Teter
Major, United States Army
M.M.A.S., U.S. Army Command & General Staff College, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

ABSTRACT

The development of an intelligent autonomous vehicle that can perform high risk missions or operate in environments too hazardous for humans has been a long standing quest of the military community. The Autonomous Platform Simulator (APS) uses the flexibility and power of realistic graphical simulation to provide a low cost testbed for the study of real-time path planning algorithms and control strategies without the commitment of resources involved in building a prototype system. It is a bridge between the theoretical study of an abstract AI path planning problem and applied research, producing concrete performance measurements under realistic conditions.

APS consists of one or more vehicle simulators, each implemented on a Silicon Graphics IRIS/4D-70GT graphics workstation. One vehicle simulator is linked with an AI agent path planner, implemented on a pair of Symbolics AI workstations using the Automated Reasoning Tool development shell.

System trials demonstrated that APS was able to achieve real-time path planning and guidance of a realistically depicted ground vehicle navigating using digitized data of actual terrain. Communication bottlenecks currently limit the ability to make direct comparisons between human and machine control but the system holds promise to fill the gap as a pre-prototype autonomous platform simulator.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

We take this opportunity to thank the people who provided assistance or inspiration. We thank Professor Robert McGhee for getting us started, keeping us going, and serving as referee. Bill Breden for providing the code to run Professor Kwak's wavefront search program with terrain slope data developed by Dennis Felhoelter. Professor Kwak provided invaluable help with the communications and wavefront search programs, for which he wrote the original code. Professor Michael Zyda for letting us follow in his wake as he pushed the leading edge of real-time computer graphics. Mark Christian for his realistic Cobra helicopter. John Yurchak, a programmer's programmer, for patiently leading us through the labyrinthine world of C and UNIX. We thank our fellow students, for the daily stimulus of working along side bright, innovative people, never too enamored of their own projects to stop and lend a hand.

Ron Ross spent untold hours explaining some of the fine points of path planning, and terrain representation. For his time, effort, and expertise we are thankful.

We especially thank Albert Wong and the rest of the Technical Support Staff, Computer Science Department, Naval Postgraduate School for their help in understanding the workings of the department computer systems.

Finally, we thank our families for providing support and understanding and keeping us nurtured with hope.

I. INTRODUCTION

A. PROBLEM STATEMENT

The development of a truly autonomous vehicle is a long sought after goal [DODSCI83, WEISN&89]. The more autonomy and intelligence such a vehicle has, the more it can replace humans for the performance of hazardous, strenuous, or repetitive tasks. Research in autonomous vehicles has largely focused on the development of control systems that totally replace human direction and move human interaction to higher levels of generalization and abstraction. Yet no broad comparison has been done of the performance of a human operator with varying levels of automated support, versus purely autonomous control. The objective of this research is to create an Autonomous Platform Simulator (APS) in order to provide a facility for such comparisons. Performance measurements, taken under varying combinations of human and AI agent control over a simulated vehicle navigating a tactical cross-country route, provide the yardstick to compare the studied modes of operation.

One of the major tasks that an autonomous vehicle must perform is to plan a path to reach its goal and then navigate along that path. There are many algorithms for calculating (planning) an optimal path based on some traversal cost criteria [RICHBG87 contains an excellent survey of path planning methods]. In the construction of an autonomous vehicle prototype, one methodology is usually chosen and then frozen by the investment in the implementation. Another aim of the APS system is to provide a means for comparing the performance of path planning algorithms in a practical setting using real world terrain data. The replacement of an actual vehicle with a realistic graphical simulation is desirable for this research because different algorithms, hybrid control configurations, and other features can be studied without the cost of building a physical system, the risk of damage to an actual vehicle, or the risk of injury to a human driver. For this research effort, the physical

vehicle and its onboard navigation and control systems are simulated on a Silicon Graphics IRIS/4D-70GT graphics workstation.

In the APS system the simulated vehicle navigates along a pre-determined path toward a known goal. The path is produced by either an AI agent or a human planner from global terrain data, such as a map, which does not contain the location of obstacles, such as minefields, which may force a deviation from the original path. Various performance measures are monitored to evaluate different combinations of autonomous and human control. The AI agent planner is implemented on a dedicated AI workstation, a Symbolics 3650. The expert system shell used in this study is ART, produced by Inference Corporation [INFRNC85].

In developing an autonomous vehicle simulation with selectable modes of vehicle control and path planning, four distinct modes of operation are implemented characterized by whether a *human operator* or *AI agent* is performing the function. The four combinations studied are:

1 - Human path planning and a human driver.

2 - Human path planning with an autopilot capable of following the calculated path.

3 - Path planning by an expert system with a human driver controlling the vehicle based on received path points.

4 - Total autonomous (hands-off) control.

The hierarchy involved in these tasks recognizes another level above the two so far discussed, that of mission planning, which designates the vehicle's final objective or goal point of the path. In APS, the output of the mission planning process is considered a given and is always entered by the human commander.

B. THESIS ORGANIZATION

The work done in this thesis breaks down into two major areas: vehicle simulation and path planning. Work done in the vehicle simulation arena was performed by

Teter. Work done in the path planning arena was performed by Shannon. The communications work was completed by both authors.

Chapter II contains an overview of previous work done in path planning, communications, and real-time vehicle simulation that relate to this study. Chapter III contains a detailed discussion of the development of the algorithms and simulator software developed during this study. This chapter also covers the simulator vehicle environment, the characteristics that allow the simulated vehicle to react realistically, and detailed discussions of path planning algorithms. Chapter IV contains discussions on the final system implementation. Chapter V examines the final APS system. Finally, Chapter VI contains the authors' views regarding the limitations of this study and possible areas for future research.

II. BACKGROUND

This chapter provides a survey of previous work in graphical vehicle simulators and path planning with special emphasis on research done at the Naval Postgraduate School that laid the foundation for the APS project.

A. VEHICLE SIMULATORS

The vehicle simulator component of the APS system evolved out of an effort to enhance the Moving Platform Simulator (MPS) [FICHTN&88], a real-time visual simulation of the Fiber Optically Guided Missile (FOGM), ground vehicles, and three dimensional terrain, which was itself the result of a continuing series of real-time visual simulations of ground, sea, and air platforms constructed by students at the Naval Postgraduate School [OLIVER&87, SMITHD&87, MCNKLE&88, WINN&89].

APS was first implemented using the Firing Platform Simulator, FPS, a close cousin of MPS. FPS was a class project which added multiple independent views and ground vehicles that could engage each other with weapons systems. Since MPS is the direct ancestor of both APS and FPS, its description provides an understanding of the context upon which the vehicle simulator was built.

1. The Moving Platform Simulator

The Moving Platform Simulator [FICHTN&88] was developed at the Naval Postgraduate School on a Silicon Graphics, Inc. IRIS 4D/70-GT graphics workstation. MPS allows a user to select a view from either a ground vehicle or FOGM missile and guide the platform over a three-dimensional view of a 10x10 kilometer area of Fort Hunter-Liggett, California. The FOGM missile is able to target, track, and destroy vehicles on the ground. The elevation data for the simulation was provided by the U. S. Army's Combat Developments Experimentation Center (CDEC) at Fort Ord, California. MPS accepts standard digital map data for other areas of the world.

Ground vehicles in MPS are controlled by dials on a peripheral input device. Control response is immediate. Changes in vehicle course and speed, for example, are effective during the next display cycle, making them essentially instantaneous. The major portions of MPS adopted unchanged for APS are the display routines, terrain representation, window manager interface, FOGM modeling, and the overall program structure.

2. The Firing Platform Simulator

The Firing Platform Simulator was a class project that enhanced the ground platform capabilities in MPS by adding a more realistic image of a tank (Figure 2-1), multiple independent viewing axes, and engagement between ground vehicle weapon systems. A set of driver's controls were added using a mouse to let the user manipulate the throttle, brakes, and steering.

3. Vehicle Motion Modeling

Realistically simulating the response of vehicles to controls, such as throttle and steering, and to changes in the terrain, is often neglected in a graphical simulation because a complete model of the physics of motion would be both analytically complex and computationally expensive. Complete modeling of the mechanics of vehicle motion is a complex proposal [BARNAC64]. Much of the modeling effort of engineers has therefore focused on analyzing a smaller subproblem such as the characteristics of a vehicle subsystem like the steering or suspension. Past work at NPS, such as that of Tan [TAN86], has studied various control algorithms for an autonomous vehicle following a curving road at constant velocity. In Tan's study vehicle mechanics were modeled by numerically integrating second order equations of motion for an idealized point mass. Numerical integration provides for accurate answers to vehicle motion equations, but at the cost of extensive computation. The modeling requirements of APS are both more general and more constrained: more general in realistically modeling the effects of control inputs and the

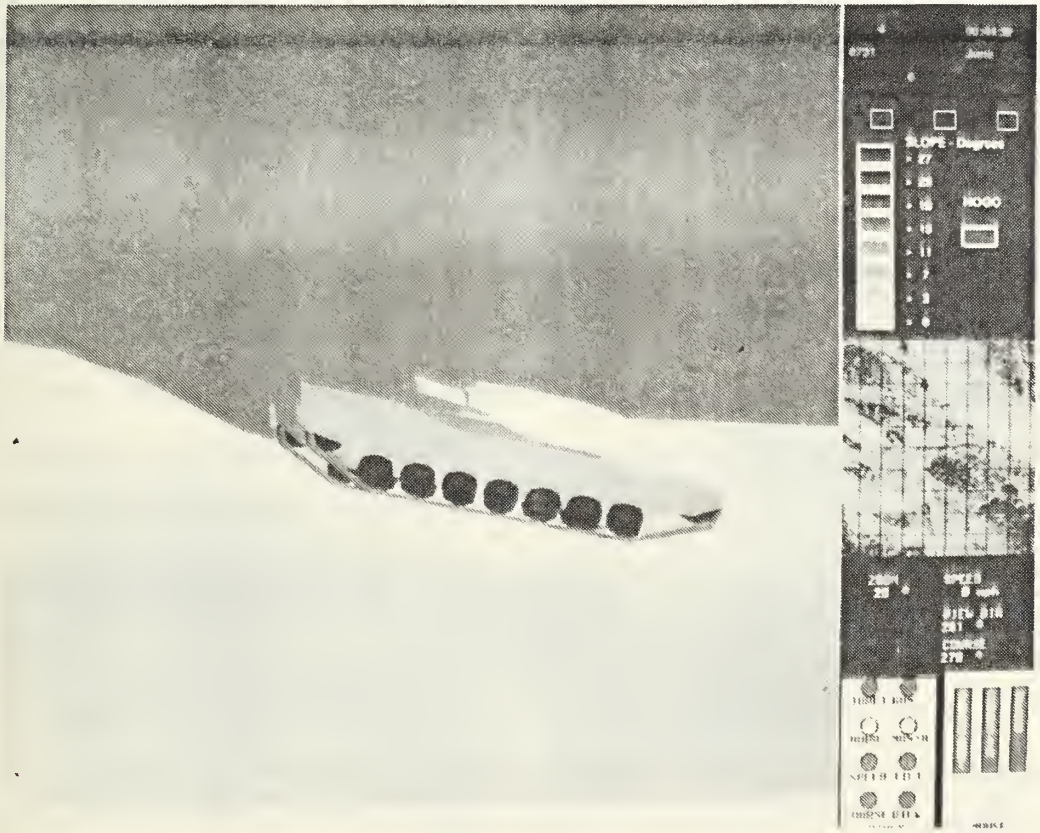

Figure 2-1 Depiction of M1 Tank

effects of vehicle interaction with varying terrain - more constrained by not being able to afford an increase in the computational burden from modeling because of the effect on overall performance.

Real-time graphical moving platform simulations often consume most of the computing resources of a graphics system in realistically depicting terrain and vehicles [FICHTN&88: pg 72]. Many graphics researchers shy away from more realistically modeling of the effects of steering and terrain in the belief that the problem is too hard and the necessary code would be too slow. What is sought for APS is a simplified model of vehicle motion and control response designed for a class of ground platforms moving off-road across varying terrain. An interesting candidate vehicle model was developed by Ross at NPS [ROSS89]. Ross's work provided a thorough but computationally simple model of vehicle-terrain interaction and energy costs while traversing varied terrain regions. Unfortunately, his model's assumption of constant velocity and its requirement for homogeneous terrain patches make it unsuitable as a basis for the APS vehicle simulator. However, his work has great potential as an alternate cost function for path planning.

B. AUTONOMOUS VEHICLES

Research in autonomous vehicles received great impetus in recent years from DoD's Strategic Computing Initiative [DODSCI83] which called for a push of "machine intelligence technology" in applied research. One of its three demonstration projects is the Autonomous Land Vehicle program. Much of the work generated on autonomous vehicles has focused on vision systems, local obstacle avoidance, such as FMC Corporation's Autonomous Vehicle [NITAO&88], and road following guidance, such as Martin Marietta's ALV [LOWRIE85]. Since APS doesn't have local obstacles to avoid or roads to follow, such research, while stimulating, lacks direct applicability. The autonomous vehicle prototypes do, however, provide insight into the functional decomposition of the problem of autonomous vehicle navigation and

control. For example, both autonomous vehicles mentioned above separate path planning from vehicle navigation and control, to the extent of having different hardware perform those functions.

The most productive source of techniques for modeling vehicle motion and response turned out to be basic physics texts such as Marion's Classical Dynamics [MARION70] or the late Richard Feynman's Physics Lecture Series [FEYNMN63]. Robotics applications [FRANK69] also provide some usable techniques. Starting then, from the solid ground of physics, albeit with several simplifying assumptions, the iterative nature of the graphics drawing loop can be used to break vehicle motion into small enough increments so that all equations of motion can be modeled with functions of no higher than first order terms and without numeric integration. More on this topic is presented in Chapter III.

C. PATH PLANNING

The task of planning a path across a known region has been classified as a weighted-region problem [RICHBG87: pg 15]. The weighted-region problem requires finding the optimal-cost path between two known points given an appropriate area-cost map. The area-cost map is described as a two dimensional region that is divided into sub-regions containing a value of traversal for each sub-region. Solving the weighted-region problem requires searching this two dimensional region. There are many strategies that can be applied to this problem of path planning. Each strategy has unique characteristics that determine its suitability. Two areas that have a major impact on path planning are terrain representation and search methods.

1. Terrain Representation

Natural terrain is generally not discrete nor clearly defined by regular boundaries. A variety of terrain models are used to depict natural terrain. The choice of terrain representation affects the choice of the search method used, and conversely

the choice of a particular search method limits the terrain representations that can be used.

a. Cartesian Grids

Regular geometric grids are used to divide the terrain into small regular cells that are used to classify some aspect of the terrain. In work done by Felhoelter, [FELHOE88: pg 36-37] slope data derived from a DMA source file of Fort Hunter-Liggett is used to classify each cell of the region. A wavefront search method can then applied to this type of region representation to find the optimal path between two points [ROWE&88: pg 2].

b. Hierarchical Models

A hierarchical terrain representation, as used by Metea and Tsai [METEA&87], is a variation of the Cartesian method used above, and is used to divide terrain into increasingly fine geometric grid cells. The lowest level contains the highest resolution data. Each cell within a level contains a single number that is associated with the cost of traversal. At the lowest level, this number is normally a direct representation of some aspect of the patch of terrain represented. At each succeeding level, the values of the cells of the preceding level that are contained in a cell of the next level are used to calculate the value of that cell.

Alternative forms of hierarchical terrain representation [KUAN84, ROSS89] move away from the Cartesian grid representation. These models group regions from a lower level that have similar representational value into larger regions at succeeding levels. The salient point of this approach is that hierarchical terrain models group terrain information from a lower level into larger regions at higher levels.

c. Homogeneous Model

Homogeneous terrain representation [ROSS89] groups contiguous points, with identical costs of traversal, within an arbitrary convex polygon. The homogeneous terrain model allows large areas of terrain to be grouped and stored efficiently in the terrain data base. It also removes the directional biases imposed on the

terrain by Cartesian terrain models. This representation is required for certain types of path planning techniques. One such technique involving Snell's law uses the principles of optics to find paths across homogeneous regions[ROWE87, ROWE&88].

2. SEARCH METHODS

The backbone of any path planner is the search algorithm used. The choice of which search algorithm to use is based on many factors. One key factor already discussed above is the terrain representation used. The following search methods are discussed briefly with emphasis on the impact of the choice of terrain model.

a. Wavefront

Wavefront planning needs a terrain model that divides the search area into uniformly sized cells, typically Cartesian grid cells, where each cell contains its cost of traversal. This technique uses a modified breadth first search where expansion is accomplished according to the cost of traversing a cell instead of simply expanding from one level of cells to another [RICHBG87]. Disadvantages to this approach are as follows:

(1) The terrain is cut up into uniform pieces no matter what the lay of the land is. This is of concern because the resolution of the search region is a direct reflection of the resolution of the cells that make up the search region.

(2) The wavefront algorithm investigated in this thesis expands to the 8 neighbors of a square grid cell, causing motion to be restricted to straight lines, in the vertical, horizontal, and diagonal directions, between a cells.

(3) Finally there is a certain amount of waste associated with the propagation of a wave. The entire wave must be expanded instead of just following the most likely path. This same problem of an ever expanding agenda is associated with a pure breadth first search.

The major advantages of this algorithm are that it is guaranteed to find the optimal path and it is well understood.

b. Depth-First

A depth-first search is used by Goodpasture [GOODPA87] to provide motion planning for a computer simulation of an autonomous walking machine. The depth-first search algorithm is guaranteed to provide a path if one exists. It however, does not guarantee finding an optimal path. The first path found is the path chosen. The algorithm used simply explores neighboring nodes that have not been explored or are not obstacles. A node is chosen that is closest to the goal. This strategy is followed until the goal is reached or the trail ends. If the trail ends, the algorithm backtracks the path, marking the used nodes as obstacles, until it finds an unexplored node to follow. If an unexplored node is found the search is continued as before. If the start point is returned to, and no unexplored nodes are available, the search fails. That is, no path is possible. The depth-first search is best used where "go" "no-go" terrain features are used.

c. A Star (A)*

The A* search combined with Snell's law is used to solve long range path planning problems, where the terrain is divided into homogeneous-cost regions. Variations of Snell's law are used to find possible paths to the goal, across the homogeneous-cost regions. Then the A* search is performed using evaluation values developed from the A* search [RICHBG87].

D. EXPERT SYSTEM SHELLS

Technology has advanced beyond the days of using a general purpose computer merely to relieve humans of the tedious tasks of redundant mathematical calculations or the endless searching of records. It is now possible to undertake more complex tasks with improved accuracy. Specifically, the growing complexity of model representation combined with a limited understanding of the processes of human thought and reasoning, have led to the use of logic oriented languages to help represent rules used in human decision making. Two such logic oriented languages

are LISP and Prolog. But these languages require the researcher to be very closely tied to the machine environment. With these languages, the programmer is directly involved in the detailed management of rules and facts. The desire to remove the burden of rule and fact management has lead in part to the development of expert system shells.

The use of expert system shells as logical programming environments provides an arena for the development of computer programs to solve problems otherwise difficult to formulate. These environments or shells provide such features as backward chaining, forward chaining, inheritance, and fact and rule management. Backward and forward chaining control strategies provide one of the critical features of expert system shells, since these strategies constitute inference engines. The ideal inference engine allows rules to fire independently of the order with which the programmer places the rules in the program control structure. Actual inference engines contained in expert system shells may fall short of this ideal, but such shells provide a tool that allows programmers to think of rules as independent islands of action waiting for the ocean of knowledge around them to provide the preconditions for their firing. The expert systems shells available at the Naval Postgraduate School are KEE by Intellicorp [INTEL86], and ART by Inference Corporation [INFRNC85].

E. COMMUNICATIONS

The real time control of a visual simulation can involve the use of more than one type of architecture. The ability to transmit and receive control information and working data between processes implemented with different architectures was investigated by Barrow [BARROW88]. The medium of communication between the various architectures was TCP/IP using the Ethernet. The principal forms of communication investigated were I/O stream and broadcast.

Broadcast datagrams were used by Barrow to communicate between IRIS workstations. They provided a convenient way to send discrete messages without

connecting hosts or requiring a specific host address. This method of communication was not supported between UNIX TCP/IP systems and the Symbolics CHAOSNET, so stream communications were used. The package of routines developed supported messages containing a single character or number with the UNIX side of the connection required to act as the connection server.

F. DEVELOPMENT SYSTEM DESCRIPTION

1. IRIS Graphics System

a. SGI IRIS/4D-70GT Graphics Workstation Description

The IRIS/4D GT is a line of high performance graphics workstations with extensive hardware support for graphics modeling that can support the real-time 3D drawing of the large number of polygons necessary in a realistic vehicle simulator [ZYDA&88]. This system has the following performance characteristics:

- 10 MIPS cpu (MIPS, Inc. R2000 RISC Processor).
- 40,000 10 X 10 pixel quadrilaterals per second (lighted & Gouraud shaded).
- 24-bit Z-buffer.
- Parallel modeling matrix operations.
- Hardware transformation matrix stack.

b. Software

The following software products were used in the development of the APS system:

- SGI C (MIPS) compiler.
- UNIX system V Operating System with TCP/IP Network extensions.
- SGI 4Sight™ Window Manager. 4Sight manages screen and I/O resources of the IRIS workstation. It supports graphics clients using the SGI graphics library as well as programs written for NeWS and XWindows[SGI4UG88]. APS runs as a client under the 4Sight server using the graphics library interface for maximum performance. This gives APS the flexibility of running in a window of arbitrary size and location. The window manager also provides the popup menu services used extensively by APS. 4Sight also provides a font

manager to scale and render text fonts in prompts, legend text and displayed messages.

2. Symbolics

a. Symbolics 3600

Symbolics 3600 workstations were chosen to perform the path planning tasks of APS. The Symbolics family of symbolic processing machines are designed with a proprietary CPU, which allows these systems to have LISP and other symbolic programming languages implemented more efficiently and effectively than conventional computers. Much of the efficiency and effectiveness of the Symbolics workstations is obtained through hardware implementation of some system management schemes. Some of the special architecture features used in the Symbolics workstations includes: tagged architecture, multiple caches, hardware stack pointers, pipelined instruction cycles, and parallel processing [SYMBOL88].

b. Software

The following software products were used in the development of the path planner for the APS system:

- Symbolics Operating System, Genera 7.1, provided a consistent background for the programming environments.
- The Automated Reasoning Tool (ART) by Inference Corporation is the principle control language for the AI Agent. This rule-based, symbolic programming language is implemented on Symbolics workstation SYM4.
- Symbolics Common LISP is used to provide access to existing path planning search algorithms and communications code.

3. Network

Computer systems in the NPS Computer Science Department are linked through an Ethernet local area network connecting some 76 stations. Average daytime traffic is 25 packets/second or 30% peak utilization in worst 20 second period[1].

[1] Based on a 24-hour test period during January 1989.

The portion of the network used by APS is shown in Figure 2-1. The vehicle simula-

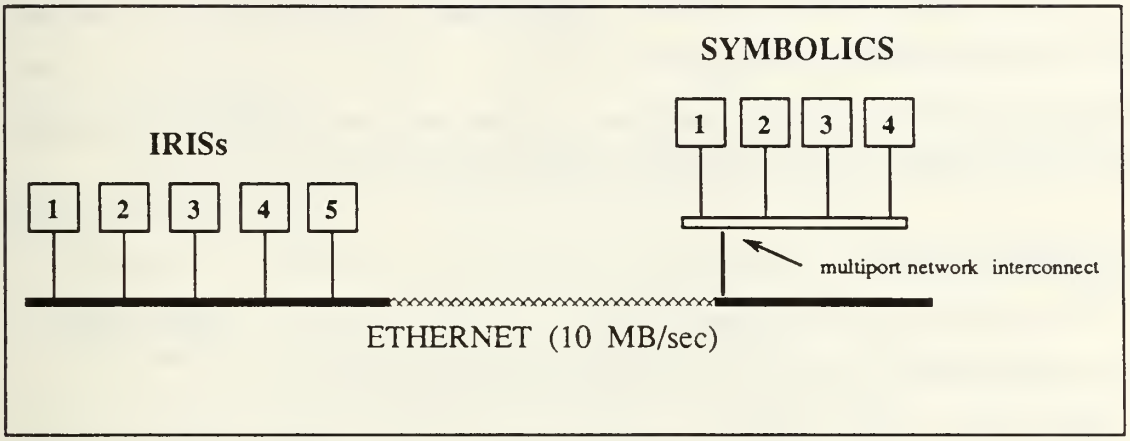

Figure 2-1 Network Physical Topology

tors (commander and driver) are connected directly to the main Ethernet cable segment. The Symbolics AI workstations are connected to the network through a mulitport network interconnect, a Digital Equipment DELNI. The flow of communications as seen by APS is shown in Figure 2-2.

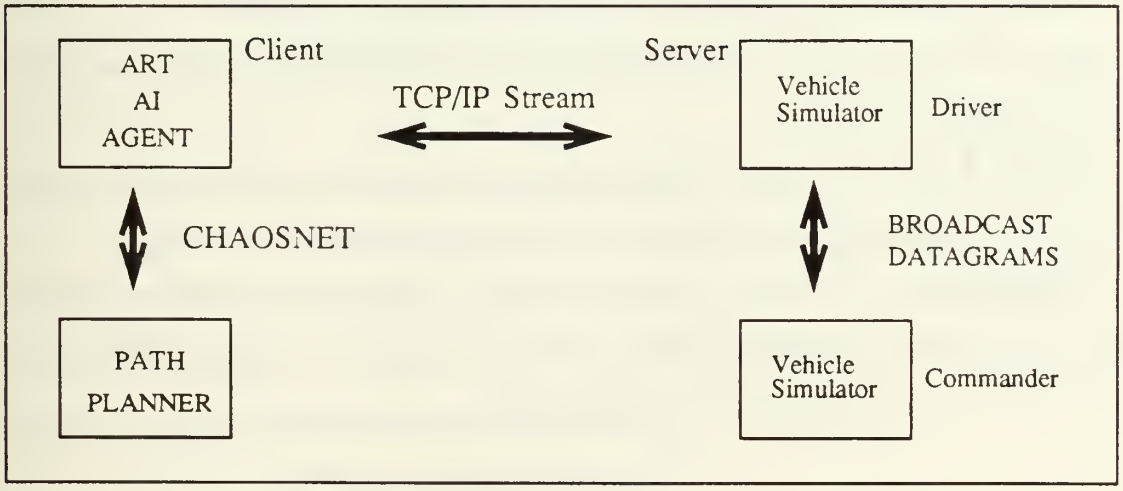

Figure 2-2 Network Logical Topology

III. METHODOLOGY AND ASSUMPTIONS

In this chapter, different candidate methodologies and algorithms are explored and the rationale behind the ones chosen discussed. The goal is to explain why certain design decisions were made and how previous work was utilized. Small segments of code or ART rules are included to show the flow from theory to application.

A. DEFINITIONS

The following terms are defined here either because they are either used in a non-standard manner or are key to the concepts presented in this thesis.

Slope Angle - The magnitude (absolute value) of the angle between the planar terrain polygon and the horizontal plane.

Local Platform - A platform added at the driver's vehicle simulator.

Net Platform - A platform added at a remote vehicle simulator and updated over the network. If a network platform is selected to operate, only the viewing controls are active. Other vehicle parameters are controlled by its home simulator.

NOGO Terrain - Terrain classified as having a trafficability of zero.

Path - A list of two or more terrain points. The first point on the path is its start, the last point is its goal.

Terrain Polygon - Planar polygon having uniform slope. In APS these are triangles, one half of a terrain grid defined by the four elevations at the vertices.

Trafficability - The relative speed at which a vehicle can traverse a class of terrain due to roughness, obstacle density, soil conditions, etc. In APS trafficability is purely a function of the magnitude of the slope angle.

B. VEHICLE SIMULATOR

The vehicle simulator portion of APS provides for a realistic depiction of a tactical platform, its control response, and its interaction with the terrain in a graphical

simulation without the overhead of completely modeling the full suite of time consuming and complex physical motion and dynamics.

1. Assumptions

The vehicles and terrain simulated in this study are assumed to have the following characteristics:

- Tracked or wheeled tactical vehicle travelling offroad, capable of traversing 60% slope.
- Trafficability of slope limits vehicle speed before stability limit is reached.
- Trafficability of slope limits vehicle speed before engine power.
- Single gear ratio modeled. (Although different gears could be modeled by using an array of time constants).

2. Coordinate Systems

The SGI graphics software library uses a three-dimensional (3D) graphics *world* coordinate system (Figure 3-1) in which the Z axis represents depth, distance from a plane perpendicular to the eye, rather than altitude or elevation. Another coordinate system is used when planning a route across terrain, corresponding to a two-dimensional (2D) view of the terrain from directly above. This is the Universal Transverse Mercator Projection, (UTM) coordinate system and is used for path planning, as in a military map, and is the coordinate system used for the terrain database. In the UTM system each point is represented by a Grid Zone Designator, a distance in meters North from the Grid Zone origin (a northing), and a distance in meters East from Grid Zone origin (an easting). The UTM coordinates of a point (x,y,z) defined in the graphics *world* coordinate system can be found by:

```
utm_x =  x + ( x_grid * 10.0 );
utm_y = -z + ( y_grid * 10.0 );
```

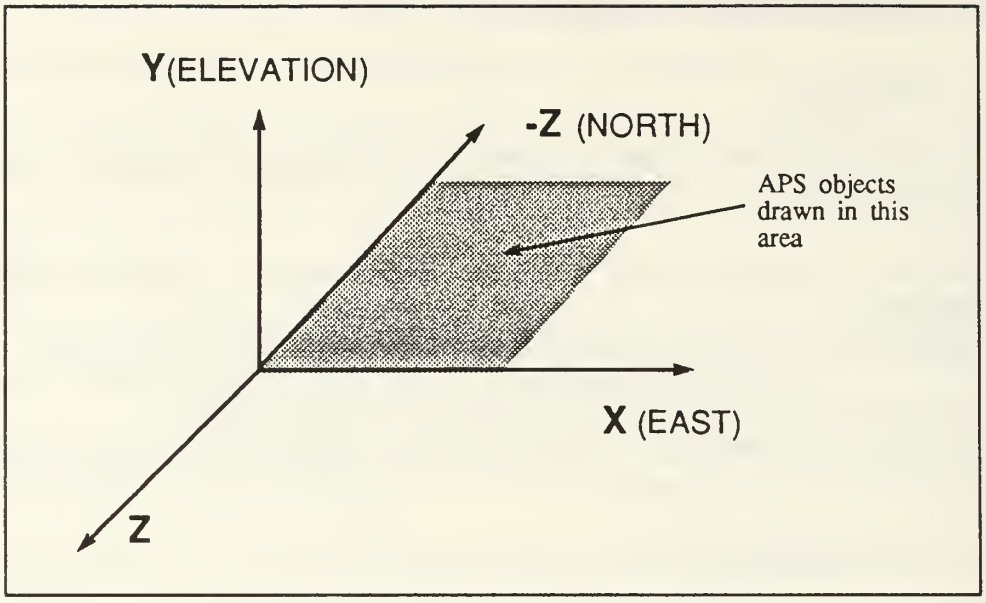

Figure 3-1 Graphics *World* Coordinate System

3. Platform Rotation Angles

In order to model moving objects, a convention must be established for the rotation of the body (platform) axes in relation to the graphics *world* axes. Normally a platform's direction or heading is given as counterclockwise degrees from North. Weapon systems such as artillery pieces are also aimed or "laid" using an *azimuth*, an angle that follows the same convention for direction but a uses a different unit of angular measure, mils (milliradians). The SGI graphics system and APS follow a different convention. Rotation angles are measured as *counterclockwise* angles from the positive axis. Thus a vehicle heading due North would have an *azimuth* (rotation about the *world* Y axis) of 1.57 radians or 90 degrees[1]. Other rotation angles follow normal right-hand rule conventions except in the case of roll. With body axes assigned as in Figure 3-2, the following conventions are established for APS:

[1]Graphics primitives use degrees but the C library functions use radians. All angles in APS are stored as radians and converted as necessary.

azimuth - Rotation about the Y-axis is in the right-hand sense, from the positive X-axis, *Counterclockwise* as an observer looks along the positive Y-axis toward the origin. Also called platform's *course* or *orientation.*

pitch - Rotation about the Z-axis is in the right-hand sense, from the positive the X-axis, *Counterclockwise* as an observer looks along the positive Z-axis toward the origin. Angle between ground (X-Z) plane and body X-axis.

roll - Rotation about the X-axis is opposite to the right-hand sense from the positive Z-axis. Here the rotation is *Clockwise* as an observer looks along the positive X-axis toward the origin.

heading - Compass course is a *Clockwise* angle in degrees between north (world minus Z-axis) and vehicle X-axis. Not used internally in the model, but it is used to display platform azimuth to the user.

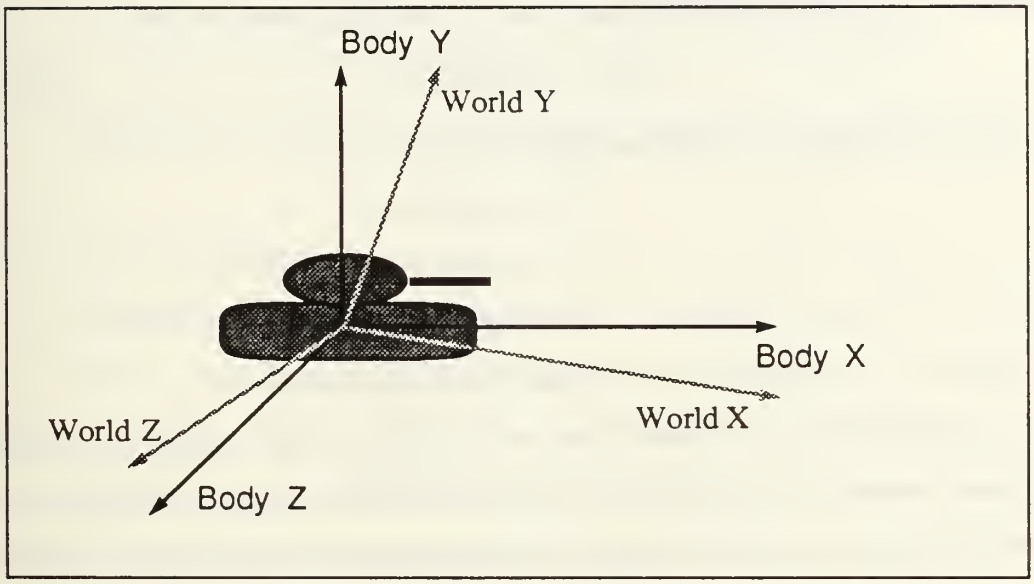

Figure 3-2 ***Body*** **vs** ***World*** **Rotation Axes**

4. Coordinate System Transformations

Since the user's viewpoint is fixed with respect to the vehicle or *body* axis, and the graphics software requires such points in terms of its own *world* unrotated axis, there is a requirement to *transform* points between coordinate systems. The position of a coordinate in a rotating coordinate system with respect to a fixed or reference coordinate system can be represented by a 3 X 3 *rotation matrix* $\mathbf{M}_{\mathbf{ROT}}$. The rotation

angles are known as "Euler" angles. The rotation matrix representing rotations about Euler angles, called *yaw* (ψ), *pitch* (θ), and *roll* (ϕ), in that order is:

$$\mathbf{M_{ROT}} = \mathbf{R_{Z,\,roll}}\ \mathbf{R_{Y,\,pitch}}\ \mathbf{R_{X,\,yaw}}\ \text{[FU\&87: pg 25]} =$$

$$\begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\psi - \cos\theta\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix} \quad (3\text{-}1)$$

The transformation of a three dimensional vector representing the body offset to the fixed reference is achieved by pre-multiplying[1] $\mathbf{M_{ROT}}$ by the vector or:

$$\mathbf{P_W} = \mathbf{P_B}\,\mathbf{M_{ROT}} \qquad (3\text{-}2)$$

This transformation requires the following operations:

3	sin function calls
3	cos function calls
16 + 9	floating point multiplications
4 + 6	floating point add/subtract

Fortunately the overhead of these operations can be avoided by solving a more general problem that includes translation and scaling during the transformation. Such a transformation from *body* to *world* coordinates is normally done by means of a 4 X 4 *homogenous transformation* matrix. This matrix represents the location of a rotated and/or translated coordinate system (*body*), with respect to a fixed coordinate system (*world*). Symbolically then, the transformation is $\hat{\mathbf{P}}_\mathbf{W} = \hat{\mathbf{P}}_\mathbf{B}\,\mathbf{M_{ROT}}$, where $\hat{\mathbf{P}}$ represents the 4 X 1 homogenous coordinate vector.

The geometry engine of the IRIS is designed to perform these type of transformations using 4 X 4 matrix operations efficiently. The *world* coordinates of

[1]Note that in graphics a body offset is transformed to where it would appear in world coordinates so the rotation matrix is pre-multiplied by the position vector. In robotics where objects actually move the rotation matrix is post-multiplied by the position vector.

the eye position vector, for example, can be calculated by having the IRIS hardware perform rotations as if the *body* were an object about to be drawn and pre-multiplying the rotation sub-matrix of the result by the offset vector. The result is the *world* coordinate offset position. Figure 3-3 is an extract of **transform_body_to_world** that performs these operations using the IRIS hardware.

```
/* Do rotations in reverse gimbal order */
rotate( (Angle)(azimuth*RTOD_X_10), 'Y' ); /* azimuth */
rotate( (Angle)(elevation*RTOD_X_10),'Z' ); /* pitch */
rotate( (Angle)(-roll*RTOD_X_10),     'X' ); /* roll */
getmatrix( offset_mx ); /* Get accumulated rotation matrix */
*eye_x = dx*offset_mx[0][0] + dy*offset_mx[1][0] +  dz*offset_mx[2][0];
*eye_y = dx*offset_mx[0][1] + dy*offset_mx[1][1] + dz*offset_mx[2][1];
*eye_z = dx*offset_mx[0][2] + dy*offset_mx[1][2] + dz*offset_mx[2][2];
```

Figure 3-3 Transforming Body Offsets to World Coordinates

5. Physics of Motion

Vehicle motion and control response is modeled as changes in the vehicles velocity vector v, with changes in its magnitude being acceleration or braking, and changes in its direction being steering. The model treats control inputs as changes to the normal constant velocity equilibrium state on level ground. The vehicle engine, at a particular throttle setting, provides sufficient force to overcome all resistance forces and maintain a certain speed corresponding to equilibrium between propelling and resistance forces. If the propelling force is increased then the vehicle will accelerate up to a new equilibrium velocity. If throttle is decreased then it will "coast" down to a new equilibrium velocity. The vehicle velocity corresponding to maximum throttle is a program constant, MAX_GNDSPEED = 45 MPH.[1]. Braking is modeled as

[1] In APS there is one set of model constants. All types of vehicles react and "feel" the same to the driver. A jeep accelerates no faster than a tank. However, these constants could fairly easily be expanded to an array of constants indexed by vehicle type.

deceleration at a variable but velocity independent rate. Steering response is modeled as an exponential function of time.

a. Friction and Coasting

A vehicle of mass m, and velocity vector $\vec{v}$, travelling on a level surface, has momentum $m\vec{v}$. At equilibrium, the only forces opposing motion are frictional resistance forces, F_R. Frictional rolling resistance is largely fluid friction and comes from air resistance, lubricant fluid resistance in bearings and gears, tire deformation while rotating, soil deformation, etc. For each resistance force

$$F_R = -km v^n \vec{v}/v \tag{3-3}$$

Different resistance forces have different exponents for v. For example, for air resistance at low speeds (< 24 meters/sec), $n \cong 1$ [MARION70: pg 53]. In fact for all resistance forces at the range of speeds dealt with in this study, $n \leq 1$ is assumed. For simplicity a convenient approximation is made that the force contribution from all sources of resistance can be combined into one resistance force with $n = 1$, with some combined constant k.

Looking at just at the magnitude of the resistance force and remembering that it is always opposite the direction of motion, (3-3) becomes:

$$F_R = ma = m dv/dt = -kmv \tag{3-4}$$

Eliminating constant mass and integrating over time this becomes:

$$\int dv/v = -k \int dt \tag{3-5}$$

which has a solution of the form:

$$\ln v = -kt + C \tag{3-6}$$

Using the initial condition $v(t=0) = v_o$ means $C = \ln v_o$. Taking the exponential of both sides gives:

$$e^{\ln v} = e^{(-kt + \ln v_o)} \tag{3-7}$$

$$v = e^{-kt} \cdot e^{\ln v_o} \tag{3-8}$$

$$v = v_o e^{-kt} \tag{3-9}$$

Let t = 1/k. Then $v = v_o e^{-kt} = v_o \cdot 1/e = v_o / 2.718$. The quantity 1/k is called a *time constant* and corresponds to the time it takes the velocity to decrease to ≈ one third of its original value. This time constant, τ, can therefore be used to calculate an average rate of change per unit time or:

$$v = v_o - (v_o \cdot dt/\tau) \tag{3-10}$$

Note that this equation depends only on the time interval and velocity at the beginning of the time interval. It also avoids calculating the exponential function. The constant τ controls how quickly the vehicle coasts to a stop or lower equilibrium speed. A large value of τ corresponds to a streamlined, wheeled vehicle on hard pavement, as opposed to a small value of τ which might represent a track laying vehicle in muddy soil. The coasting function (3-10) is coded as: **coast_vel = currvel - (currvel / COASTING_TIMECONSTANT * elapsedsec);**.

An analysis of a typical case shows how well this code fragment produces the same result as equation (3-9). For **COASTING_TIMECONSTANT** = 10.0, **elapsedsec** = 0.5, and **currvel** = 40 MPH, after 10 seconds elapsed time, (3-9) yields 14.72 MPH while the code produces 14.34 MPH. In APS the final velocity for coasting need not be zero. It could be a lower equilibrium velocity. The exponential nature of the decrease in velocity means that the new velocity would be approached asymptotically, never actually reaching the target velocity (variable **cmdvel**). Therefore there is a cutoff in the procedure **velocity_model**, to wit:

```
If ( fabs( cmdvel - currvel ) < TO_MPS ) return( cmdvel);
```

This returns the selected velocity as the current velocity if it is already within 1 MPH.

b. Braking

Deceleration due to braking can be modeled as a variable resistance force that is independent of velocity. For a braking factor b, $0.0 \le b \le 1.0$,

$$F_{\mathbf{BRAKE}} = ma = m\, b\, dv/dt = -k_{\mathbf{BRAKE}}\, m\, b \tag{3-11}$$

Eliminating constant mass and rearranging, the new velocity is given by:

$$v = v_o + dv = v_o + (-k_{\mathbf{BRAKE}}\, b\, dt) \tag{3-12}$$

or, in code:

```
newvel = currvel + ( MAX_DECELERATION * brake_factor  * elapsedsec );
```

where **MAX_DECELERATION** is a constant representing the maximum rate of deceleration before skidding (shear force between vehicle and ground > frictional force) and **brake_factor** represents the input to the model from the vehicle controls, a value of 0.0 representing no braking down to -1.0 or 100% braking. This control input can come from dials, the mouse or be calculated by an autopilot.

c. Acceleration

Acceleration corresponds to advancing the throttle position to a new velocity position, v_T, causing engine output to exceed the propelling force necessary to overcome the current rolling resistance. It assumes linear engine power output. Subtracting equilibrium forces at the current velocity, v_o, gives:

$$F_A = m\, dv/dt = k_A(\, v_T - v_o\,)\, m \tag{3-13}$$

$$dv = k_A(\, v_T - v_o\,)\, dt \tag{3-14}$$

$$dv = 1/\tau\,(\, v_T - v_o\,)\, dt \tag{3-15}$$

Where τ is again a time constant = $1 / k_A$. Equation (3-15) is implemented as:

```
newvel = currvel + ( ( cmdvel - currvel ) * dt / ACCELERATION_TIMECONSTANT;
```

For **ACCELERATION_TIMECONSTANT** = 10 seconds and **cmdvel** = 40 MPH, this gives:

$0 \Longrightarrow$ 20 MPH in 6 seconds

$0 \Longrightarrow$ 30 MPH in 11 seconds

$0 \Longrightarrow$ 40 MPH in 21 seconds

This compares well with the nominal acceleration for the US M1 Tank of $0 \Longrightarrow 20$ MPH in 7 seconds [JANES87: pg 122].

d. Slope Calculations

Elevation is a function of UTM coordinates, *h(x,y)*. The gradient of *h* is a vector in the ground (X-Z) plane that points in the direction of greatest **increase** of altitude.

$$\nabla h = \left[\begin{array}{cc} \partial y / \partial x, & \partial y / \partial z \end{array} \right] \tag{3-16}$$

This model is not so concerned with the direction of the gradient vector as with its magnitude and the magnitude of the *slope angle* ψ_h which is the angle between a tangent to the elevation function and the X-Z plane.

$$\psi_h = \tan^{-1} \left[\begin{array}{c} (\partial y / \partial x)^2 + (\partial y / \partial z)^2 \end{array} \right]^{1/2} \tag{3-17}$$

Fortunately, it is not necessary to calculate the slope angle directly. It can be calculated from the terrain polygon patch surface normal unit vector **N** which is already available for each terrain polygon from the lighting model calculations. Call ψ_N the angle between **N** and the X-Z plane, which is also the map ground plane. Since **N**, with components x_N, y_N, z_N, is perpendicular to the terrain polygon, $| \psi_N | + | \psi_h | = \pi / 2$. Now $\psi_N = \tan^{-1} (y_N / (x_N + z_N)^{-1/2})$, and since $\tan(\pi / 2 - \theta) = \cot(\theta)$ and $\tan(\theta) = 1 / \cot(\theta)$, then

$$\psi_h = \tan^{-1} ((x_N + z_N)^{-1/2} / y_N) \tag{3-18}$$

In the vehicle simulator, this result is produced by the function **convert_normal_to_slope** which returns ψ_h in radians.

If the *surface normal* of the terrain polygon is not readily available (perhaps because vertex normals are being used), the *effective slope* of the vehicle can be calculated from its pitch and roll. These body angles are used to calculate the world coordinates of a *body normal*, a normal vector which points out the roof of the vehicle, using another math function **transform_body_to_world**, shown in Figure 3-4.

```
float  normal[3], slope;
transform_body_to_world( platform->cse,
                         platform->base_pitch,
                         platform->base_roll,
                         0.0, 1.0, 0.0,
                         &normal[0],
                         &normal[1],
                         &normal[2] );
slope = convert_normal_to_slope( normal );
```

Figure 3-4 Calculating Effective Slope Using Platform Orientation

Finally, vehicle pitch alone can be used as effective slope to limit speed, since, stability factors aside, pitch angle is the slope resistance the engine must overcome.

e. Effects of Slope

Instead of directly modeling the effect of acceleration forces on a vehicle due to gravity when travelling on sloping terrain, the model assumes that the *trafficability* of sloping terrain limits vehicle speed before engine power would be insufficient to maintain a set speed. Maximum vehicle speed is limited by a slope governor that decreases maximum speed as a function of slope. This slope governor

function is shown in Figure 3-5 where MAX_GNDSPEED is the maximum speed a

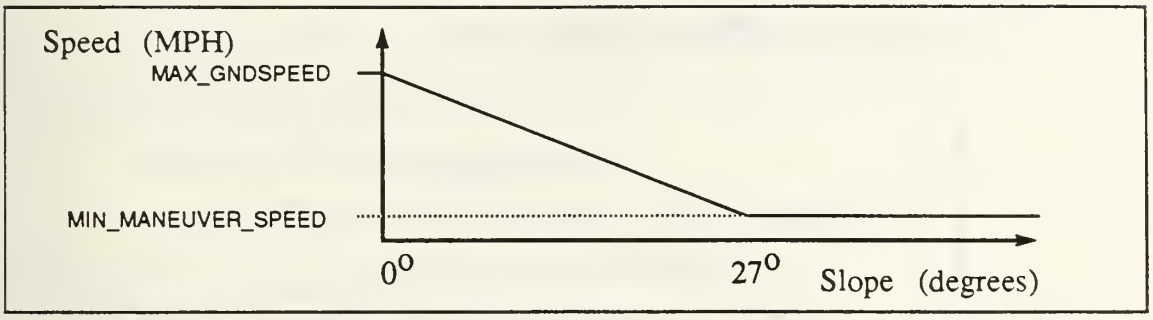

Figure 3-5 Slope Governor Function

vehicle can achieve on level ground and MAXSLOPE (27°) is the maximum slope traversable by the vehicle. Since limited speed could go to zero in untrafficable terrain, the vehicle would be stuck, unmovable, if it ever entered a **NOGO** terrain patch. A low "maneuver" speed is allowed for the driver to carefully and slowly work his way out of such a situation.

f. Suspension Oscillation - "Bounce"

When a vehicle crosses a bump or other change in terrain slope, it induces an oscillation in the spring-mass suspension system. This oscillation continues until it is damped by the shock absorbers and friction of the vehicle suspension moving parts. This motion can be quite difficult to model due to the complex geometry of a vehicle suspension system. However, by limiting oscillations to changes in vehicle pitch angle only, this motion is easily modeled as simple damped harmonic oscillation along a single axis (the vehicle pitch angle) as shown in Figure 3-6. This *transient pitch* is then added to the base vehicle pitch caused by the

slope of the terrain. Two additional fields in the vehicle data structure are created to handle this effect, **bounce_amplitude** and **bounce_time**.

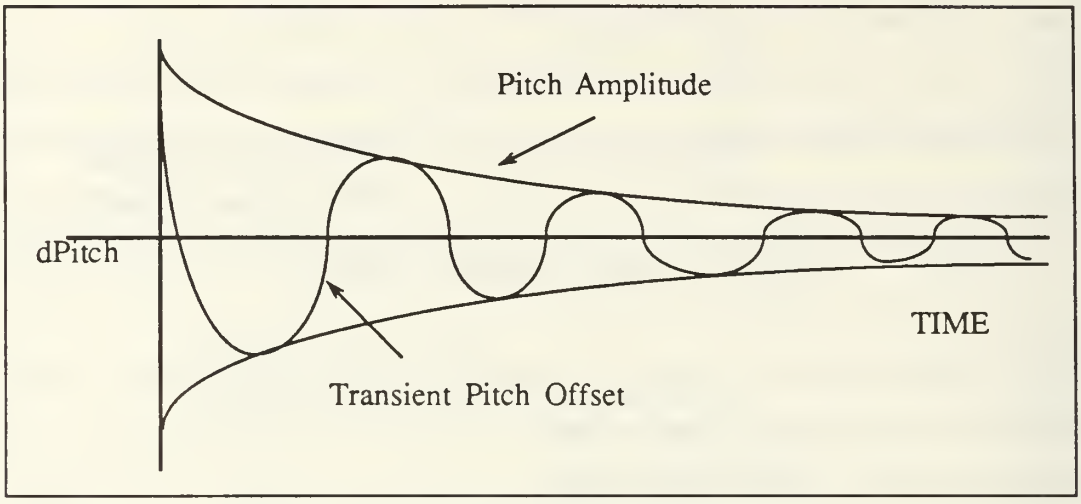

Figure 3-6 Damped "Bounce" Oscillations

The equation for damped harmonic motion [MARION70 :pg 371] is:

$$y = y_o \, e^{-\beta t} \cos(\omega_D t + \delta) \qquad (3\text{-}19)$$

$$\Downarrow \qquad \Downarrow$$

amplitude oscillation

where $\beta = b / 2m$, b is the damping force, m is the mass of the vehicle, and ω_o is the frequency of *undamped* oscillations of the system. Assume $\omega_D \cong \omega_o$. Choose a time constant τ, which is equal to the time interval when amplitude of the oscillations has decreased to $1 / e$ of its original value, which is $\tau = 2m / b$. Equation (3-19) can then be written as:

$$y = [y_o - (y_o * dt / t)] * \cos(w_D t) \qquad (3\text{-}20)$$

If the displacement y is the suspension travel at the front wheels then $\theta = \tan^{-1}(y / \text{wheelbase})$. For relatively small displacements, the damped harmonic oscillations can be calculated directly using the pitch angle θ and (3-20) becomes:

$$\theta = [\theta_o - (\theta_o * dt / \tau)] * \cos(\omega_D t) \quad (3\text{-}21)$$

This can be broken up into two code steps for each cycle of the update loop:

1) Calculate the current angular oscillation value.

```
bounce_pitch = bounce_amplitude *
    cos( OSCILLATION_FREQUENCY * TWOPI * total_time );
```

2) Calculate new bounce amplitude based on damping effect.

```
bounce_amplitude = bounce_amplitude -
    ( bounce_amplitude * dt / DAMPING_TIMECONSTANT );
```

With **DAMPING_TIMECONSTANT** set to (1 - 1/*e*) / τ.

Empirically, equation (3-21) and its code can be shown to approximate the results of (3-19). Considering a typical case[1] and comparing just the decline in amplitude after 3.0 seconds, equation (3-21) yields 5.5° (converted from y displacement), while the code gives 4.8°. Only a small part of this difference (app 0.1°) comes from oscillating the pitch angle directly instead of oscillating the displacement and then converting, 5.5° versus 5.6°. The total error is small enough that "tuning" the constants can bound this error well within the difference detectable in a moving visual simulation.

6. Simulation Time Interval

The model time interval, or ***dt***, using Leibnitz notation, is the elapsed time required to complete one processing loop in simulation time. Since the rates of change of most of the processes are non-linear, the linear approximation used is only a good approximation if ***dt*** is small, « 1 second. The second problem that results if ***dt*** is not small enough comes from delayed control feedback. For example, if steering a vehicle in a turn and ***dt*** is of the order of 1 second then the driver will tend to overshoot control corrections, making it difficult to steer onto a desired course or avoid obstacles. This lower bound is due to control response and depends on many factors,

[1] For DAMPING_CONSTANT = 3.0, dt = 0.25 seconds, wheelbase = 2.0 meters, initial displacement of 5 meters ≅ 15 degrees, and total time = 3.0 seconds.

including platform velocity, control responsiveness, complexity of maneuvers, etc. Responsiveness for an aircraft travelling at hundreds of MPH must be greater than for a ground tactical vehicle travelling cross country at speeds typically < 25 MPH. For such ground vehicles, a subjective lower bound appears to be 3-4 frames or cycles per second.

7. Paths

Paths in APS resemble the military concept of a "route" with an SP (Start Point), RP (Release Point) or goal, and a CP (Check Point) at turning points or critical points. Figure 3-7 shows the path data structure. The SP of the path is its first point and the RP is the last point on the path. Each platform data structure (Appendix A) contains a pointer to a path and a pointer to the next point along the path. Path manipulation routines are contained in module **path.c**.

Paths are created and maintained separately from platforms. When a vehicle is "assigned" a path to traverse, a copy of the path is made for the platform and a pointer to the platform is added to the list of platforms assigned to that path. Thus several platforms can be assigned to traverse a path and navigate along it independently. Also, if a point on the original path is altered, all affected platforms can be notified. On the other hand, if a platform must deviate from the path to avoid an obstacle, intermediate points can be inserted in the platform's copy of the path without affecting the original.

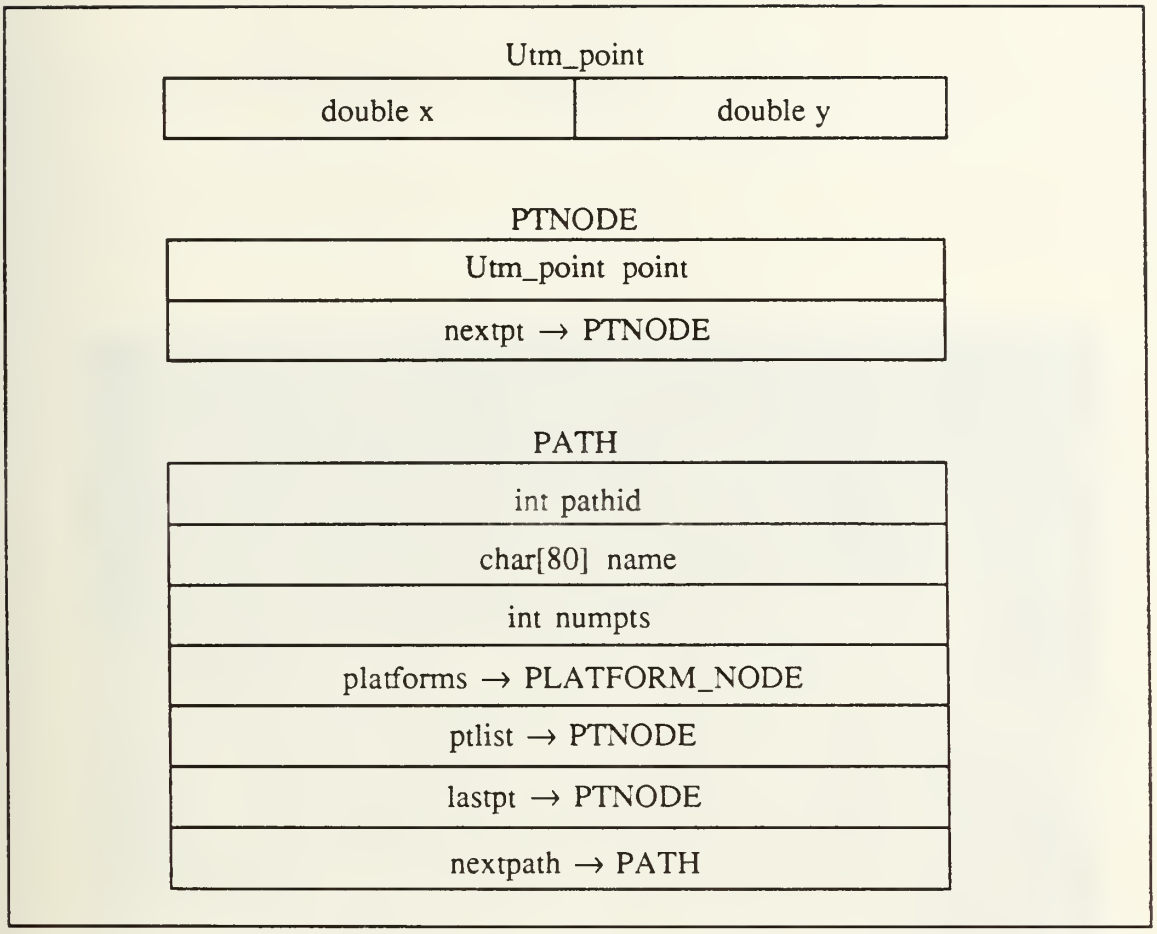

Figure 3-7 Path Format

A platform being guided by an external agent by receiving one point at a time is actually following a path that consists solely of a periodically updated goal. As new guide points are received, the goal point is replaced. The autopilot module can then navigate a platform along a path by heading successively toward each point on the path list. Figure 3-8 shows a tank approaching the goal point, which is drawn as a tall, pyramid marker on the terrain. The paths are maintained as a linked list managed using four global variables:

pathlist - pointer to first path on path list.

pathlistend - pointer to last path on path list.

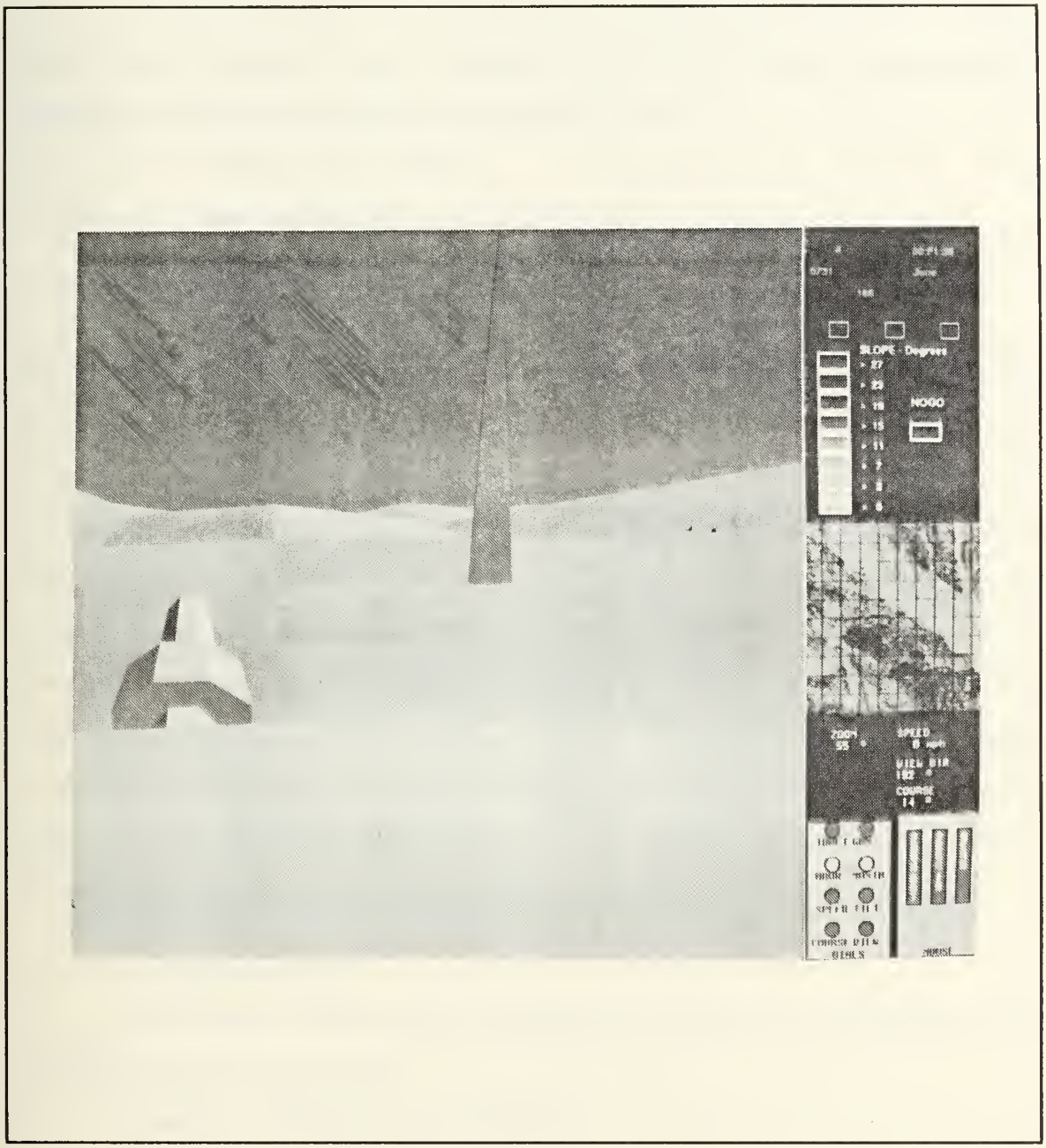

Figure 3-8 Driving a Tank Toward Its Goal

numpaths - number of paths currently on pathlist.

path_pickid - unique identifier for graphics picking.

Path manipulation is accomplished by selecting the "PATH OPERATIONS" entry in the main menu. The path operations menu is then constructed and displayed, providing for the selection of up to four functions:

1 - Display Paths - ON/OFF

2 - Construct a Path

3 - Delete a Path

4 - Assign Vehicle to a Path

The first option toggles the display of paths on the 2D terrain map. Figure 3-9 shows the display used to manipulate paths. Paths are displayed by default but may be turned off to reduce screen clutter. Menu option 3 is only presented if there is at least one path defined. Option 4 is only presented under the additional condition that at least one platform is defined. At present, all platforms except FOGM can be assigned to a path. A path, once defined, is stored in a file containing all currently defined paths. When APS is started, it searches for a file "aps_paths.dat" in the following directories, in order: the current (default) directory, the directory containing the APS executable, and the "DTED" directory. If the file is found, APS loads the paths it contains. Each time a path on the path list is created or destroyed, the file is updated.

8. Guidance States

The current control state of each platform is reflected by the combination of two fields in each platform record:

control - MANUAL or AUTOPILOT

ext_guidance - ON or OFF

The slot **ext_guidance** determines whether platform guide points are taken from incoming message or an assigned path. The slot **control** determines whether course

Figure 3-9 Terrain Map Used for Path Planning

commands are calculated by the autopilot or are provided from the vehicle controls (dials or the mouse joystick).

9. Autopilot

The autopilot determines the commanded course and speed for each local platform that has its control field set to AUTOPILOT and has a path defined. Since external guidance messages update the platform's local path record, the autopilot functions irrespective of the source of the path data. The autopilot calculates an azimuth to the current guide point. The current guide point is automatically updated to its successor on the path (if there is one) when the platform is within VICINITY meters of the way point. If the platform gets within ARRIVED_DISTANCE of the guide point then the guide point must be the last point on the path, i.e. the path *goal.* If so then the autopilot applies the brakes to bring the platform to a halt without overrunning the goal. Figure 3-10 shows the relationship of these distances. Precise

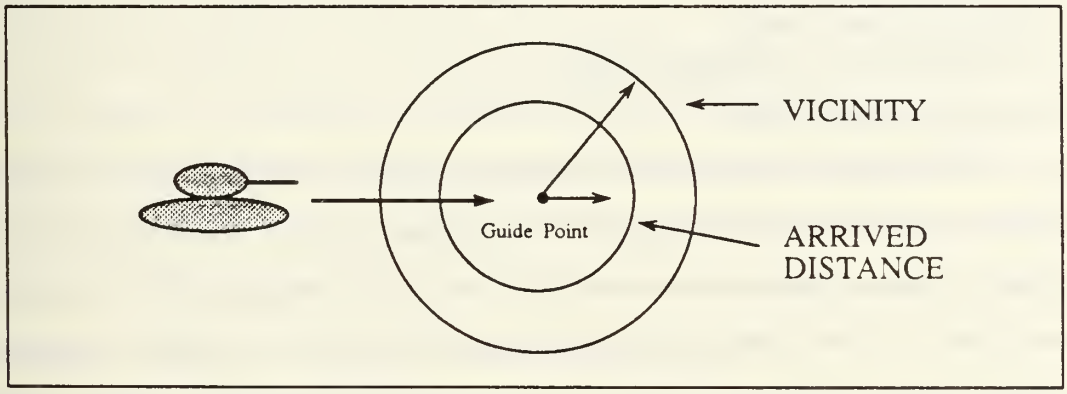

Figure 3-10 Autopilot Control

control would require both these distances be variables that are a function of the platform's speed, instead of constants, so that a platform starts to brake or turn in the time necessary to stop or turn exactly over the guide point. This level of precision would be necessary to navigate a platform through a field of obstacles and would require projecting ahead the platform's location so as to issue the proper commands at

the correct time. However, the simpler algorithm presently used is adequate to halt a platform travelling at maximum ground speed (MAX_GNDSPEED) before reaching the goal.

B. PATH PLANNING

1. Introduction

The path planning process for this study simulates the actions of the vehicle commander in planning paths, and issuing waypoints for a controlled vehicle. The commander starts out with the following facts:

- Which vehicle is being controlled.
- Start point and goal locations are known.
- A terrain map of the region to be traversed is available.
- The terrain map contains cost of traversal information for the region.
- Vehicle speed.
- Vehicle course.
- Vehicle guidance mode.
- Vehicle location in UTM coordinates.
- Current simulation time.

The commander uses this information to plan a path to the goal, selecting the quickest path between the start point and the goal point. In selecting a path, the commander chooses prominent terrain features as guiding waypoints for the driver. Once the path is selected, the commander issues commands to the driver to proceed to the first waypoint as indicated by the commander. The commander continues to issue new waypoints as the driver pilots the vehicle close to the last waypoint. As the simulation continues, the commander needs to be informed of changes to the vehicles status as indicated here.

- Vehicle ID.
- Vehicle speed.
- Vehicle course.
- Vehicle guidance mode.

- Vehicle location in UTM coordinates.
- Current waypoint location in UTM coordinates.

2. Path Planner Control Program

The path planning simulation of the commander is divided into two major areas, the overall controlling program and the actual search algorithm that does the path planning. The term *path planner* is used to describe the combined AI processes that make up the AI simulation of the path planning commander. The path planner is kept separate from the graphics simulation of the vehicle and implemented on the Symbolics AI workstations. Two reasons for this are: first, a great deal of path planning work done at the Naval Postgraduate School is done using AI workstations, and second, a substantial amount of the program code produced is done in LISP and Prolog that can be easily ported between the different AI workstations.

The path planner control program is separated from the actual search portion of the path planner for three reasons. The first is to allow modularization of the code. The communication costs associated with this approach do not appear to override the benefits of being able to substitute different search algorithms. The second reason for the separation was to allow the path planner control program the exclusive use of a workstation. Expert system shells require considerable system resources, and it is felt that the overhead of running the expert system shell would put an excessive load on a single workstation when combined with running real time path planning searches. Finally, this separation should allow more than one search program to work simultaneously.

a. The Expert System Shell

High turnover and short learning curves predominate much of the frustration associated with thesis work. Therefore, since one of the major goals in this study was to provide a test platform that could be used to study the relationships associated with the application of artificial intelligence techniques to the control of autonomous vehicles, it would be advantageous to have the path planner controller

written in a high level symbolic programming language. The use of such a language allows a researcher to examine problems at a much higher level of abstraction than with LISP or Prolog.

One of the criteria in the selection of an expert system shell was the desire to have the path planner control program continuously monitor the knowledge database and react to changes therein. Forward chaining control strategies facilitate this continuous flow within the rule based system by simply keeping fresh facts asserted. In ART and KEE, the forward chaining mechanism is self contained in the inference engine of the shell. There are forward chaining control implementations written for both LISP [MCNKLE&88] and Prolog [ROWE88], but abstracting the researcher away from the mechanics of forward chaining produces code which is easier to understand.

ART was chosen over KEE in part because it appeared easier to examine the workings of the rules in ART. ART allows direct manipulation of the rules through Symbolics' ZMACS editor, and through the use of ART's ability to watch and record the firing of rules, and the assertion and retraction of facts.

b. How ART Works as a Process Controller

ART is a rule-based, expert system shell, containing the ability to forward chain and backward chain. The principle inference engine is the forward chainer. As stated in Chapter II, an ideal inference engine within a shell would provide an uncluttered view of the rules and knowledge base used in a problem. In reality, there are inherent limits on the inference engine implemented within ART. One important limit is that an artificial structure and order are imposed on rule firing. A simple example of this is the difference between the firing of two rules that require identical preconditions. One rule must fire first. The engine must decide. The choice could be as simple as choose the rule that appears first in the program structure, or choose the shortest rule. And though the choice may be arbitrary it must be consistent. ART appears to choose the first rule in the program structure.

(1) Rule Structure. ART's rule structure provides a straightforward way of declaring the complete predicate logic for a given rule. A sample rule extracted from the path planner control program is presented in Figure 3-11 below. The left side of the rule contains the preconditions necessary for the rule to fire. The right side of the rule carries out actions. These actions can be controlled by binding temporary facts and by examining states through the use of conditional statements. The parts of the rule are clearly shown in Figure 3-11. Here the rule is fired when the fact **(menu one)** is asserted. The right side of the rule can request the operator to perform some action

```
(defrule MENU1
(schema sym
        (one ?s1)
        (two ?s2)
        (three ?s3))
?a <- (menu one)
=>
(printout t t "Where is the path planner located?")
(printout t t "Your choices are the following, chose one by it's letter. "
          t "a " ?s1
          t "b " ?s2
          t "c " ?s3
          t "NOTE---Please ensure that the path planning software is running"
          t)
(bind ?b (read))
(if (or (eq ?b 'a)
        (eq ?B 'A))
    then
      (assert (sym-link ?s1)
              (menu two)
    else
      (retract ?a)
      (assert (menu one)) ) )
)
(retract ?a)
)
```

Figure 3-11 MENU1 Code Fragment

such as choose the Symbolics machine where the search control program resides. The operator's response is then checked to ensure a valid response, and facts are asserted that enable the Symbolics communications start up rule to fire and start

communications with the appropriate Symbolics workstation. Of special note here is that a fact in ART must be retracted before it is reasserted. If the fact is asserted before it is retracted, the assertion will be lost. This is because ART keeps only one copy of identical facts. The Path Planner Control Program contained in Appendix B provides a more detailed look at the code.

(2) Continuous Forward Chaining. A rule firing control mechanism is needed that allows the path planner control program to continuously monitor the knowledge base and the communications sub-process. This is accomplished with continuous forward chaining by cycling through a base set of rules. The rules chosen for this cycling are the interface with the vehicle clock and the calls to the system's communications ports. These rules are selected because they are the most likely routes for new facts to enter the knowledge base of the commander. The Symbolics **read-char-no-hang** stream read method is used in the communications rules to ensure that the communications calls do not wait if there is no data on the lines. Rule cycling begins when a rule has met all of its enabling preconditions. A rule fires when all of its enabling facts are met. The rule **check-comm-links-iris** retracts its enabling fact, and asserts the facts **(check-comm iris)**, **(check-comm sym)**, and **(clock-update yes)**. Order of assertion is important here because the last fact asserted will be pursued first, as explained in Paragraph c. below. If no information is available from the IRIS communications link, the clock is updated, the Symbolics' communications link is checked, and finally ART cycles back through the checking of the IRIS communications link.

(3) Rule Precedence. Actions by a human commander are taken according to some precedence or order imposed by the commander's judgement. It is desired to duplicate the human's ability to judge and separate actions that need to happen immediately from those that could be postponed. Assigning rules an order of precedence allows more important rules to be examined first. Rule precedence is accomplished by the use of ART's salience function. Salience values are from -10000

(lowest precedence) to 10000 (highest precedence). A rules salience value is assigned at compile time. If the rule's salience is not declared, a value of 0 is assigned. Rules of the same salience are loaded onto a stack as they become ready to fire. Rules thus grouped are fired according to their salience value first, then according to their position on the stack. It is useful to think of ART as having a separate stack for each salience value, and always firing rules off the stake with the highest salience value. At each level of precedence, the rule loaded to the stack last is fired first. This provides a mechanism to mimic the human ability to pursue what should be done first. Rules must be written such that more important things have higher saliences than less important things. The consequences of this stack action effectively imposes a most recent fact following algorithm. A side effect of a most recent fact following algorithm is that it can lead to indefinite postponement of rules. This can happen in two different ways. First, if high precedence rules are continually added to the agenda stacks, low precedence rules will never fire. A second and more subtle way a rule could be indefinitely postponed is by asserting a fact that activates a rule of the same precedence as the postponed rule. Since all newly asserted rules are loaded to a stack, the most recent rule is looked at first, thus postponing the older rule. This indefinite postponement is easily handled in Prolog by using an **assertz** command. Using ART, the programmer must control rule firing by sequentially activating rules, and ensuring that all sequences of rule firings lead back to the lowest level.

(4) LISP Calls. LISP calls are used where it is more convenient to perform an action on LISP data structures or to use existing LISP functions. LISP calls can be made only on the right hand side of rules, and are delineated by #L immediately before the LISP code. ART can make direct use of LISP symbols and values, but is clumsy at manipulating LISP lists. Therefore, LISP lists are converted to ART facts and schemas that use relations within ART to link related facts. An example of this, in Appendix B, is the rule **process-waypoints**. In this rule, the

incoming waypoints are stored with the vehicle and their sequence number from the list they came from. ART also fails to recognize the strings produced by calls to the LISP communications packages. Here Common LISP provides the **intern** function to convert a string into a symbol. This symbol is then fed to ART. As can be seen in Figure 3-12 below, the **intern** command requires a prefix that designates which Symbolics package the LISP function is defined in. The ART package does not have all of the Symbolics' Common LISP functions available.

```
(bind ?b #L(scl:intern (scl:send talk-i :check-iris 3)))
(if (eq ?b '>>>) then
```

Figure 3-12 Code Excerpt from the Rule read-update

3. Path Planning and Search Algorithms

The second portion of the path planner is the search algorithm. The requirements for the algorithm are that it accept as input the following data:

- Start point
- Goal point
- Vehicle ID
- UTM coordinates of the lower left hand corner of the 10 KM grid window.

The output requirements are waypoints passed individually with the corresponding sequence number and vehicle ID.

a. Search Region Representation

Planning a path across real or simulated terrain requires some criterion be established that will allow the path planner to choose between routes. Slope is a common terrain feature used as a simple distinguishing factor [ROWE&88]. The greater the slope, the greater the cost of traversal. This criterion has some interesting properties that are not in accord with the physical environment. In APS, *effective slope* is an absolute value independent of the direction of traversal. In traversing an

actual physical region a given incline has varying degrees of relative slope depending on the traversal angle. Since the major goal in this study is to build a test platform that would allow the testing of search algorithms and their interfaces with simulated vehicles, this discrepancy is accepted in the interest of simplifying the problem. An interesting result of this simplification is that bidirectional searches can be performed, because the cost of traversal is independent of the direction of travel across a given region with a given slope.

Discrete geometric cells were used because of the simplicity of conversion from the graphics elevation data to the slope data used by the wavefront path planner. The use of the wavefront search technique was based on the construction of the elevation and slope data files and the ease of implementation of the wavefront algorithm. The slope data files produced by Felhoelter's methods were designed to contain all of the information about a given search region [FELHOE88]. This information included the boundary information that ensured the search algorithm would not overflow the search region. This boundary information was otherwise unrelated to slope information of the region. The stripping off of this boundary information was trivial and could be accomplished while building the slope files or after they were complete. However it became apparent that the use of slope data files containing one by one to ten by ten kilometers of slope data would prove difficult. Using files built in this manner would have required the use of 1225 to 625 separate slope data files for a 35 KM by 35 KM map that covered the same region as the map used by the IRIS based vehicle simulator. It would have also required either predefining the area to be searched or some other way of selecting the appropriate file for a given run. Initially a 35 KM by 35 KM slope data file was used, but it was found that the time to read in the data took as long as six minutes. This long read-in time occurred because each record of the file had to be read in sequentially until all of the data for a given map was read in. This read-in time was reduced to one minute by converting the text file to a binary file and using the Symbolics LISP **file-position**

function, which is Symbolics' equivalent to the **lseek** function of C. Finally, the slope files were recomputed using the graphical methods developed on the IRIS workstations. This was done because the methods used by Felhoelter produced significantly different slope data than that produced on the IRIS workstations. This difference appears to be based on the fact that the vehicle simulator uses the slope calculated from the lower left triangle of a one hundred meter square in the graphics simulation. These triangles were used because they form the planar surfaces used in the graphical displays on the IRIS workstations, and the drawing routines provide normals to the surface from which slope can be easily calculated. These methods were described earlier in this chapter. The only significant difference between the two methods is when the slope calculations are performed. The slope information for the Symbolics processes is calculated before the simulation is begun, while the slope information is produced at system run-time by the vehicle simulator.

C. AUTONOMOUS vs. MANUAL CONTROL

The guidance and control states of APS have been previously described. What follows contains a fuller explanation of what is involved in the transition between these states. The states were designed to be as independent and flexible as possible, to allow switching in and out of autopilot control while being guided by an external agent and, conversely, to allow switching external guidance on and off while remaining under autopilot or manual control. Ideally, the source of external guidance, human or AI agent, would be transparent to the guidance system. Unfortunately, the methodology used for human control introduced asymmetries into the design. An externally guided vehicle is controlled by a remote human path planner on another graphics workstation differently than it is guided by the remote AI agent. The human commander designates a path for a remote platform vehicle just as if it were a local platform. The path is then transmitted across the network in its entirety. Thereafter the vehicle driver navigates as if the path had been generated locally. On the other

hand, the AI agent transmits one path point (guide point) at a time, successively updating them as the vehicle gets near. The source of this lack of symmetry lies in the greater functionality of the AI agent. It was designed to calculate a new path if the controlled vehicle encountered unforeseen obstacles or deviated too far from the calculated path. This cannot be done in advance. This dual role of global and local planner was never envisioned for the remote human commander except in the case of replacing one global path with a new one. In essence then, the external guidance state becomes one of exclusive AI agent control and the methods used for the transitions back and forth between external and internal guidance are designed to accommodate the different models of guidance and preserve the transparency to the rest of the vehicle simulator.

A platform's external guidance can be toggled ON or OFF either locally by a popup menu selection from the driving menu or remotely by network message. This network message is generated by a remote human commander making the same menu selection as would the local operator. If the selection is made locally, the mode transition is made. If the selection is made remotely, the message is transmitted. Actions on the local platform are the same regardless of the source of the command. At present, no authentication or permission system is used, nor is there a local lockout or override provided.

External Guidance OFF --> ON causes the following actions:

1) Set **ext_guidance** toggle ON.

2) Send an INITIALIZE control message to the AI agent containing the UTM coordinates in meters of the origin of the current ten kilometer box, vehicle identifier, start and goal points of the path, and current simulation time (If no AI agent is connected the message is discarded). Note that the start point sent is the platform's current guide point which may not be the SP of the originally assigned path. If the platform had partially navigated a path under internal guidance, then the guide point will be the next point in the remainder of the path.

3) Set up the platform to receive guide points by making the platform's current location its guide point and deleting the remainder of its path. This is done so that the autopilot, if engaged, will simply bring the platform to a stop instead of heading out directly for the goal. The portion of the path traversed so far is preserved on the front of the list.

4) Finally a position update message is sent over the network on both broadcast and stream channels. Currently it is this **UPDATE** message, with its guidance field set to ON, which triggers the AI agent to calculate an optimal path based on global terrain cost data. However, there is nothing in the vehicle simulator to prevent the AI agent from choosing the start and goal points by itself, sending a message to turn guidance ON, and then sending guide points from a calculated path.

External Guidance ON --> OFF causes the following actions:

1) The platform's external guidance flag is set to OFF. This causes any incoming guide point messages for this platform to be ignored.

2) The platform's path is deleted.

3) Its original assigned path, if any, is reloaded.

4) The platform's guide point is set to the point on the original path closest to the platform's current location. In this way, a platform taken off external guidance after navigating a portion of a path would not go all the way back to the start point, but can complete the remainder of the path. Note that the closest path point is not necessarily the best path point pick to minimize travel time or some other performance measure. Locating the best path point is a non-trivial problem in itself. In some cases the simple method used will guide the platform back to a previously passed path point or directly to the goal point. Generally however, when assigned a path with many fairly short segments, backtracking and loss of time will be limited to one half the length of the current path segment.

5) Finally, a position update message is sent over both broadcast and stream channels. The guidance field of this message reflects its new state and directs the AI agent to stop sending guide points.

D. COMMUNICATIONS

The amount and sequence of data that must be passed over the network is determined by the functions to be performed. For communications between vehicle simulators, sufficient data is needed to display the platform on a remote simulator as well as model its movement. Information flow with the AI agent is determined by the division of labor between the vehicle simulator and the commander, human or machine. Updates are sent between vehicle simulators or to the AI agent only when a state variable such as speed, course, weapon firing, etc., changes. In general, in communicating with other vehicle simulators on the network, the vehicle simulators PUSH information over the network using broadcast datagrams to any others who might be listening. Only upon initialization does the system poll for a response.

There are usually several methods to chose from when communicating between applications over a LAN. In the case of the APS development environment, TCP/IP supports byte streams, which require dedicated connections, and datagrams, which may be connectionless, and even addressless in the case of broadcast datagrams, or may be sent between connected hosts. The vehicle simulator's use of a PUSH broadcast system to communicate with other vehicle simulators is adequate for the amount and types of messages needed by that portion of APS. It would have been simpler if this same approach could have been used for communicating with the AI agent. Broadcast datagrams provide for reliable[1] transmission of discrete messages over a LAN. This means that specific addresses need not be hard coded or determined at

[1]Datagrams are not usually considered "reliable" because there is no receiver acknowledgement. If communication between hosts is entirely intra-network then the underlying protocol, in this case Ethernet's CSMA/CD, guarantees delivery to each host and, barring buffer overflow or process termination, the message will reach each process properly attached to the addressed port.

run time and that each read will return zero or one complete discrete message (provided the message fits within the network maximum size). However, this proved not to be feasible primarily due to the limitations of the Symbolics' implementation of support for TCP/IP network services. The only arrangement that worked during this research was a pair of halfduplex stream connections, with the further limitation that the vehicle simulator must act as the server and the Symbolics as the client. For consistency, communications between AI processors also use stream connections.

The only remaining design decision for the vehicle simulator end of the communications link was then whether to have the simulator poll the incoming stream connection for input using a non-blocking read or to spawn a sub-process to continuously monitor the connection and communicate with the main simulator process through semaphores and a shared memory buffer to hold messages.

A separate subprocess carries the additional complexity of implementing semaphores and shared memory plus the computational overhead of a context switch. Also, during development, when system aborts are common, special care must be taken not to leave orphan subprocesses when the main process terminates. The main advantages are immediate response to incoming messages and message preprocessing. The subprocess issues a normal blocking read on the connection, which sleeps until input is present. This is more efficient than constantly polling. The second plus is the ability to respond immediately to some query while the main processes may be tied up in computation and graphics processing.

The polling approach is simpler. In fact, in APS even the initial acceptance of a connection request is done by polling. On a single processor system, one CPU still must run both the main and subprocesses so no real time is being gained by running them in parallel. There may be some concern that the input buffer may overflow between polls, which in APS happen once each drawing cycle. However, under UNIX, the receive buffer can be made practically as large as desired (currently 40K bytes) or at least as large as the shared memory buffer is likely to be, so the risk of overflow

between cycles is the same. The system network daemon basically does the same job as the subprocess, and hopefully, it is more efficient at it than user written code would be. Tests using dummy AI agent programs which send messages at ten times the normal rate have not produced evidence of a lost message.

Communications with the Symbolics AI agent are performed as follows:

1) Upon initialization, the vehicle simulator establishes a stream socket, sets it to non-blocking operations, increases its receive buffer size, and creates a connection queue as a stream server.

2) Thereafter, during each graphics cycle, the socket is polled by issuing a non-blocking ACCEPT command. If a stream client, in this case the Symbolics, is waiting for a connection, two stream sockets are cloned, one for receiving messages from the Symbolics and a separate one for sending messages to it.

3) If a working connection is established, then a non-blocking read is issued on the receive stream socket. Messages from the AI agent, comprised of character strings with punctuation character delimiters are extracted from the stream and returned as whole messages to the simulator which takes the appropriate action. The specifics of how this is implemented are contained in Chapter IV. If the stream connection is broken by the AI agent, then a flag is set and the system returns to polling for a connection instead of polling for data to read.

At the Symbolics AI agent, the path planner needs to monitor the progress of the vehicle, independent from the vehicle simulator updates. This means that calls to the communications system can not be allowed to wait for data. For this reason communications at the Symbolics AI agent is done using the **read-char-no-hang** method to read the input stream. This allows reads from the I/O stream to return **nil**.

Messages are identified and delineated by non alphanumeric characters. Non alphanumeric message delimiters were chosen to reduce the chance of processing partial messages. This could occur if the first part of a message were lost over the network. It is assumed that a properly delineated message is complete and correct.

The use of a data stream requires that the formats of the messages be known in advance, and that each message be identified as to type. This is accomplished by the use of non alphanumeric delimiters as mentioned above. A further precaution that ensures messages are not lost forever should one message arrive without its leading delimiters is the use of different length delimiters on the front and back of messages. The front delimiter is longer than the back delimiter. This is done to prevent a message that has lost its front delimiter from starting a cycle of reads that could pass over the correct first delimiter. The algorithm that receives the messages on the Symbolics workstations checks for the first delimiters, and then reads in a prespecified number of characters, based on the message type. The last few characters make up the ending delimiter. It should be noted that the sending process supplies a null character between messages. If the ending delimiter were the same length as, or longer than the beginning delimiter, the ending delimiter could be interpreted as the beginning. Since the rest of the message is not evaluated until the ending delimiter is checked, message traffic could remain out of synchronization indefinitely once broken.

E. PERFORMANCE MEASURES

In order to make quantitative comparisons among path planning algorithms or human-machine control arrangements, some numerical figure of merit must be chosen. For tactical vehicles travelling cross-country, some candidates are: transit time, fuel consumption, enemy exposure, weapons line-of-sight, etc. In this study transit time was chosen because it can be tied directly to the terrain data base and platform characteristics.

For each platform experiment or trial, there is a global planning time and a transit time. In a sense, the planning time represents a fixed investment cost and transit time operating cost. An experiment may compare total time (planning and transit time) or analyze them separately. As an example of the type of trade-off study that might be made, consider the current path planning algorithm used. Such a

wavefront or breadth-first algorithm may not represent the fastest way to produce an optimal global path. However, its nature as a neighbor-based algorithm means that each path step is calculated only on LOCAL cost data. Then, assuming the agenda is preserved, when a small piece of the data changes, such as the discovery of an obstacle, only a small region need be recalculated. Its overall performance in the presence of constantly changing local data might be superior.

This research makes no attempt to produce a definitive measure of effectiveness. Rather a mechanism is sought that will provide a basis of comparison for others to use in measuring the effectiveness of path planning systems.

F. SUMMARY

This chapter provides an examination of the source, thought process, and evolution of the design of APS. The development of the vehicle motion model and control response of the vehicle simulator are discussed along with the knowledge base of the rule-based path planner and path planning algorithms. This chapter concentrated on the *why*. The next chapter will delineate the *how*.

IV. SYSTEM DESCRIPTION

This chapter describes how the methodology and algorithms were implemented, including the function and structure of some of the main programs and rule sets. Data structure definitions along with some code listings are contained in Appendix A.

A. TERRAIN DATABASE

APS uses terrain data that is a subset of a vegetation and elevation database in 12.5 meter increments for an area of Ft Hunter Liggett, California, provided by CDEC. This database is preprocessed into 100 meter resolution data by sampling every eighth point and then stored in a separate file that is read by APS. Each data point is 16 bits (2 bytes). The 3 most significant bits form a vegetation code which is used to color terrain polygons in a shade of green for the 3D view. If the vegetation code indicates light or no vegetation, or no vegetation data is available, then the terrain polygon is colored according to its elevation using the currently designated color map, usually shades of brown. The remaining 13 bits contain the elevation in feet. This elevation is used to draw the 3D terrain, calculate normals for the lighting model, and calculate slope used by the path planning cost function.

APS is currently limited to the 35 KM by 35 KM area for which preprocessed data is available. In UTM 10 meter grid coordinates, this area extends from 10SFQ41006000 to 10SFQ77009500. A basic terrain surface patch is formed by the four elevations of the vertices of a 100 meter square. These points are not necessarily planar. Since the IRIS cannot quickly render filled non planer polygons, this polygon is divided along a NW to SE diagonal into two planar triangles which are rendered as filled shaded triangles. This basic terrain patch is shown in Figure 4-1.

The lower left (SW) triangle is called the *lower* triangle and the upper right (NE) triangle is the *upper* triangle.

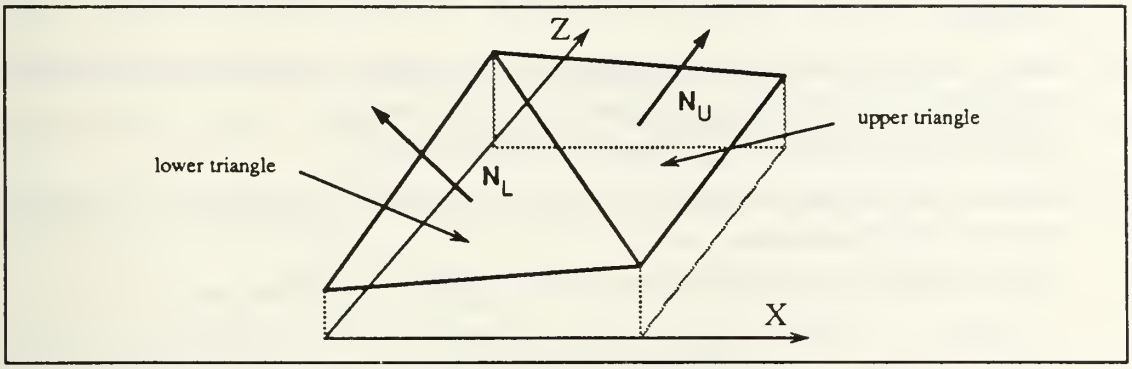

Figure 4 - 1 Terrain Patch

The elevation of each triangle vertex is stored, along with its X and Z offsets in a floating point array (the upper left and lower right points are duplicated) consisting of 72 bytes (3 X 6 X 4bytes) per 100 meter square. In addition, a surface normal 3D vector is calculated and stored for each triangle. One square kilometer of terrain data thus consumes 9600 bytes (10 X 10 X 96bytes). The entire 35 KM by 35 KM area consumes over 11 Mbytes of memory. To maintain performance, only the vertex and normal data for a 10 KM by 10 KM area selected by the user are kept in memory.

B. VEHICLE SIMULATION

1. Capabilities

The capabilities of the Autonomous Vehicle Simulator include:

- Acceleration due to changes in engine throttle (thrust).
- Deceleration due to coasting and braking.
- Change in vehicle pitch due to acceleration or braking proportional to the magnitude of the change of velocity.
- Vehicle roll due to centrifugal force while turning.
- Linear steering controls with exponential steering response.
- Damped vehicle oscillations due to changes in vehicle pitch as vehicle travels over varying terrain.
- Change in vehicle velocity due to terrain slope.

- An autopilot that will navigate a platform along a designated path.
- The ability to handle vehicle control inputs from either local driver controls, local/remote autopilot steering commands or remote autopilot/commander path commands.
- Models multiple vehicles, with selectable independent views from each vehicle representing weapon sights, commander's station view, etc.
- Multiple independent viewing axis and viewing positions.
- Multiple independent weapon system axis maintained to provide for stabilized weapon/sighting systems.
- Utilizes graphics hardware for fast coordinate system transformations.
- The ability to sight, range, and fire weapon systems, including stabilized weapon systems. The following platforms and weapons are implemented: tank with main gun SABOT and HEAT rounds, open jeep, closed top jeep, TOW jeep, truck, Cobra attack helicopter with TOW weapon, and FOGM.
- ANSI C standard source code.
- Broadcast networking to allow multiple simulations to operate together on different IRIS workstations.

A complete discussion of the user interface for APS can be found in Appendix C.

2. APS Environment

The vehicle control and motion model requires an interface with its simulator environment in four areas: maintenance of and access to terrain data structures; timing; control inputs; and display of results. From the terrain data structures, it needs the elevation of an arbitrary point in world coordinates. This is provided by the function **gnd_level**. It also requires the surface normals for each terrain polygon. Timing is provided by the routines in module **simtime** (Appendix A). Control inputs are provided by reading the mouse position, reading dial positions, or receiving commands from a remote guidance system. Displaying the results, which after all is the main thrust of the simulation, is accomplished by **drawterrain** after the model "positions" the vehicle for drawing and sets up the viewing parameters and the projection transformation.

3. Graphics Drawing Cycle

Typically, window-based graphics programs operate in a *drawing cycle* with an INPUT-UPDATE-DISPLAY loop. A representation for this cycle in the vehicle simulator is shown in Figure 4-2. The platform modeling routines operate in the update portion of this cycle. They operate on the platform data structure which is then

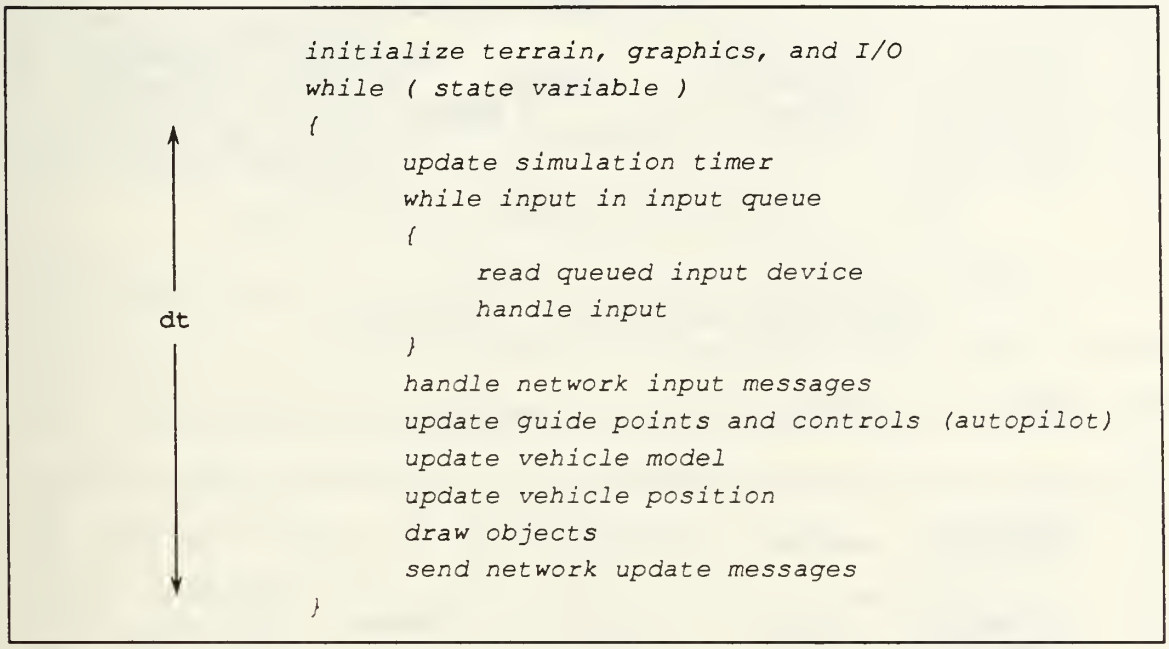

Figure 4 - 2 Structure of Main Drawing Loop in *event_driving*

passed on to the display cycle. The only parameters usually required for model routines are a pointer to the platform and the elapsed time since the last update cycle was completed.

4. Input

As discussed earlier, control inputs can have several sources, can be set to override each other, and can be turned on or off depending on the internal state of the simulator. The source of control inputs is largely irrelevant to the design of the motion model except for steering. Two ways of modeling steering correspond to two types of physical control systems. In one, the steering wheel or control device is

directly connected to the wheels, tracks or control surfaces of the vehicle (Figure 4-3). External course commands then must be processed into signals to a servomechanism which physically moves the steering control just as a human operator would manipulate it.

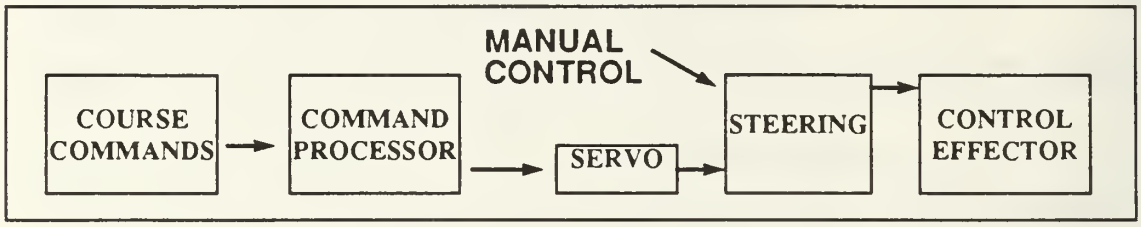

Figure 4 - 3 Manual and Automatic Steering Control

Another arrangement is "fly-by-wire" (Figure 4-4) where manual control generates a signal which is perhaps one of several input signals to a steering control system which in turn activates physical control surfaces such as wheels, tracks, or ai-

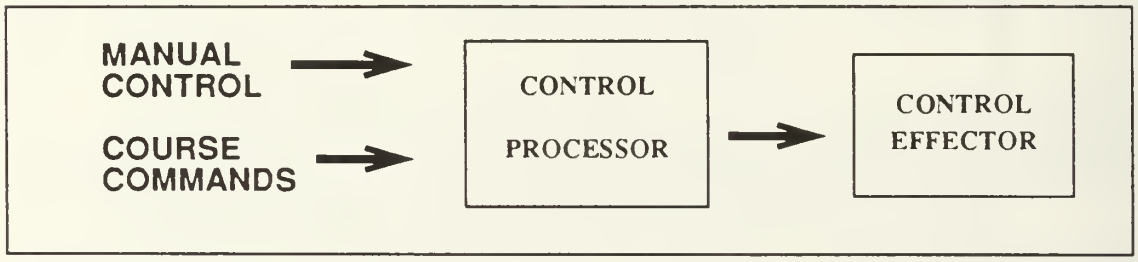

Figure 4 - 4 "Fly-by-Wire" Steering Control

lerons.

APS currently uses the first system. Course commands are converted into a turnrate. This turnrate is then used by model routines **steering_model** and **turning_model** without caring if it came from the steering wheel or remote commands. Thus the modeling of turning is independent of the source of the turning commands.

5. Model Update

The update phase (Figure 4-5) is actually split into two sub-phases. In the first phase, the new *velocity* and *course* are calculated. In addition, any transient pitch or roll caused by a change in velocity is calculated.

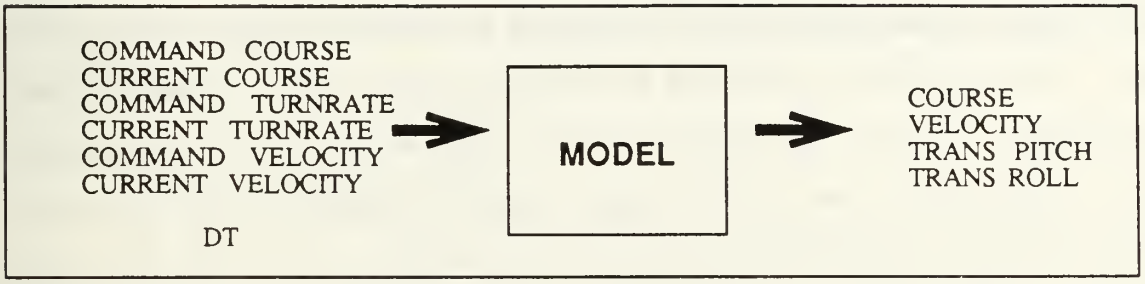

Figure 4 - 5 Vehicle Model Update Phase

Once the model has updated the platform data structure, it is passed on to the routine **update_veh_pos** which "moves" the platform to its new location and calculates orientation angles based on the slope of the terrain. Any oscillations or "bounce" in vehicle transient pitch angle is calculated by **handle_bounce**. This is based on the change in vehicle base pitch angle exceeding some threshold or minimum change. At this point, an interplay exists between attempting to smooth abrupt pitch changes between adjoining terrain patches and simulating bounce. Because the terrain is represented by patches, the flat tops of hills or ridges that are less wide than the terrain cell size are "missed" by the data base. Consequently cresting a hill and going down the other side is portrayed as an instantaneous change from positive to negative slope as the line separating the two adjoining patches is crossed. To smooth out this sharp transition the length of the baseline to the front of the vehicle used to calculate base pitch was extended forward about 20 meters. This results in the pitch change being spread over smaller increments as the reference point moves down the far slope as the vehicle is coming up the near side. Unfortunately this smoothing can also "smooth" oscillations out of existence. Only experimentation

with the constants **pitchbase_distance** and **bounce_threshold** can produce a realistic compromise.

6. Platform Position and Viewing Parameter Update

APS updates the vehicle position and orientation variables whether or not the vehicle is currently selected as the viewing platform, the ***driven*** vehicle. In MPS the viewer's position was not fixed with respect to the driven vehicle coordinate system. It was a constant **Y** offset from the vehicle's graphical center in *world*, not *body* coordinates. On fairly level terrain this works well, but as the vehicle pitches and rolls when travelling over rough or sloping terrain the viewpoint or eye position appears to bounce around inside the vehicle. This movement is disorienting and in some cases may even result in viewing the terrain from underneath the terrain polygons. One solution to this problem is simply to not draw the driven vehicle. However, the viewer then looses the frame of reference the vehicle outline provides, especially when the view angle is not directly to the front. A more satisfactory solution is to define the viewpoint as an offset from the vehicle origin in vehicle *(body)* coordinates and transform the viewpoint into the graphics (*world*) coordinates required to establish the viewing perspective. Such a transformation also allows the viewpoint to be placed at an arbitrary point in the vehicle which could represent, the gunner's sight, commander's cupola, etc. Setting the viewing perspective is then done as shown in Figure 4-6 where **eye_x, y, z** is the sum of vehicle position coordinates and the viewing point offset in transformed *world* coordinates.

```
perspective( fov, 1.0, 0.1, MAXLOOKDIST );
lookat( eye_x, eye_y, eye_z,
        local_px, local_py, local_pz, (Angle)(viewroll*RTOD_X_10) );
```

Figure 4 - 6 Setting Projection Parameters

The IRIS graphics software also requires a viewing "target" (**local_px, y, z**), for the **lookat** perspective routine. The homogeneous transform again provides a

means for calculating this visual target since it is simply a constant displacement along the body X-axis of the viewer. This corrects simplifications in MPS that neglect *cant* or *body roll* in determining point of view. This precision becomes important when a weapon system is modeled because the point of view is also the point of aim. A one degree error in azimuth caused by *cant* corresponds to a 18 meter error at a range of 1000 meters. Therefore, the MPS routine **update_look_pos** was modified to use this procedure.

7. Network Communications

As described in Chapter III, communications among vehicle simulators is handled differently than communications with the AI agent. Messages among vehicle simulators, whether each is functioning as a peer or a remote human commander, are passed over the network using broadcast datagrams, while the vehicle simulator and the AI agent communicate using a stream.

Communication routines are divided into two levels: the APS message level and the network service level. These levels and the modules that contain level routines are:

APS message level	check_for_packets (receive) network (send)
message-stream management	network_IO
network system services	netstream_services broadcast_services

a. Vehicle Simulator Communications

(1) Initialization. Two network sockets, a transmit socket and a receive socket, are initialized for each vehicle simulator. The receive socket is bound to an address containing the APS broadcast port number. This port number is arbitrary, but must be unique to avoid interference with other network services such as "mail" and "rwho" and must be the same for all vehicle simulators. In APS, the broadcast port number is a program constant (DEFAULT_BRDCAST_PORT in

network.h). An alternative method of assigning a port is by defining a "service" in the network system file "/etc/services". The system service **getservbyname** can then be used to determine the port number at run time. This method has the advantage of allowing changes in the port number without recompiling the program should the port assignment interfere with some other network application. However, each system running a vehicle simulator must have the *same* service definition for APS. Finally, each socket is set to BROADCAST mode, non-blocking I/O, and has its buffer size increased to RECV_BUFSIZE (currently 40K bytes). Since the most common broadcast message is an UPDATE packet with a size of 180 bytes, each vehicle simulator can normally receive ≈ 220 packets before the buffer overflows. In communication tests, no such loss of message traffic has yet been observed.

After the sockets are established, each vehicle simulator sends a polling message to synchronize itself with any other already running simulators. A response to this initialization message sets the initial 10 KM terrain box to that area already being viewed by a running simulator.

(2) Sending Messages. Messages are sent as character strings divided into a header string that identifies the type of message and a varying length data string. The first character in the header string is a message token, a character that uniquely determines the message type. The rest of a message is the formatted output of a **sprint** command containing from one to thirteen fields. All messages are built and sent by routines in module **network()**. This routine is called with one argument indicating the type of message that is requested. Message types are shown in TABLE 1. The function **network()** also contains several local static variables which contain data used in building a message. For example, to send an UPDATE message the routine **set_cntlmsg_platform(platform_pointer)** is called to set the

vehicle then **network(SEND_UPDATE_PACKET)** is invoked to build and dispatch the message.

TABLE 4-1 VEHICLE SIMULATOR MESSAGE TYPES

TYPE	FIELDS	DESCRIPTION
INIT_MESSAGE		Polls for other vehicle simulators.
ANS_MESSAGE	x_grid, y_grid	Answers INIT_MESSAGE and sets origin of 10KM box.
UPDATE_PACKET	vehicle id, type, UTM x,y, course, speed, weapon azimuth, weapon elevation, transient pitch, transient roll, control mode, external guidance.	Updates platform data on remote vehicle simulators.
END_PACKET	base id number	Tells remote vehicle simulators to delete all platforms belonging to this host.
FIRE_MESSAGE	firer x,y,z, target x,y,z, weapon azimuth, weapon elevation	Sends a weapon system firing event. The flight of the projectile is then modeled on each simulator.
LOCK_ON_MESSAGE LOCK_OFF_MESSAGE	vehicle id	Sends id of platform that is/is not being tracked by FOGM.
DESTROY_MESSAGE CRASH_MESSAGE	vehicle id	Sends message notifying remote simulators that platform has been destroyed.

(3) Receiving Messages. Since datagrams contain discrete APS messages, the type of an incoming message is determined by matching the first character of the header string with a character token. The data string is then disassembled by a formatted string read (**sscanf** in C) and the appropriate action taken. This message handling occurs in module **check_for_packets()**. Once during each

drawing loop this routine is called. It loops handling messages until no input is available.

b. Vehicle Simulator - AI Agent Communications

(1) Initialization. The communications stream is set up by initializing a stream socket, setting it to non-blocking I/O, increasing its buffer size to RECV_BUFSIZE bytes, and establishing a connection queue by calling the **listen** system service. The socket is then polled once each drawing cycle using a non-blocking **accept** system service. Two sockets are then cloned to handle the receive and transmit streams and a global flag, **control_connected**, is set indicating a connection with the AI agent has been established.

(2) Sending Messages. Messages are sent as a continuous string of characters with no imbedded "white space" characters such as spaces, tabs, or linefeed. All numeric fields must be zero filled. Each message is preceded by a fixed number of a unique delimiter token characters, usually a punctuation character, {,?,@, etc.. The variable length data string follows. A message is terminated by a number of delimiter characters, one less than at the front of the message. Since stream I/O implies an unbroken flow of data, these front and read delimiter characters serve to identify the type of message and provide begin and end message markers at the program level. This additional framing allows resynchronization should a portion of a message be lost or garbled. It also allow recognition of different type messages by a simple finite state machine. Messages to the AI agent are built and sent by calling

routine **control_message(message_type)** contained in the module **network**. The types of messages sent to the AI agent are contained in Table 2:

TABLE 4-2 VEHICLE SIMULATOR to AI AGENT MESSAGE TYPES

TYPE	FIELDS	DESCRIPTION
INITIALIZE	UTM x,y of 10KM box origin, vehicle id, path start x,y goal x,y, simulation time	Tells AI agent which platform to plan path for and what part of terrain database to load.
UPDATE	vehicle id, vehicle location UTM x,y, simulation time, guidance flag	Updates platform data. Initial guidance ON message triggers path calculation.
OBSTACLE	obstacle vertices	Sends coordinates of vertices of detected obstacle.
CONTROL	vehicle id, simulation time, control code	Sends status and control flags.

(3) Receiving Messages. Routine **check_for_packets()** also handles incoming stream messages by calling **recv_control_message()**. If no AI agent is connected, it polls for a connection. If it can form a valid APS message from characters in an internal buffer, then it returns TRUE otherwise it returns FALSE. Messages are recognized using a finite state machine. Incoming characters are returned to **recv_control_message()** by **get_msgchar()**, which returns the next character in the block buffer and keeps the buffer filled as necessary by reading the stream. If the stream read returns 0, then the client has broken the connection so the

current message, if any, is discarded and the flag is set to begin polling for a reconnection. Message types received from the AI agent are contained in Table 4-3.

TABLE 4-3 AI AGENT to VEHICLE SIMULATOR MESSAGE TYPES

TYPE	FIELDS	DESCRIPTION
GUIDEPT	vehicle id, path point UTM x,y	If external guidance is set for platform matching vehicle id, then the platform's guide point is replaced by the incoming point. If recv_path is set for platform then incoming point is added to path being built.
CONTROL	vehicle id, simulation time, control code	Turns guidance ON/OFF, recv_path ON/OFF, or autopilot ON/OFF.

8. Simulation Time

It is often desirable to change the speed at which the simulation runs by modifying the ratio between real (clock) time and *simulation* time. This is done by filtering calls to the system clock through the *simtime* module. This allows, for example, simulation time to be suspended while menus are displayed. The routines available are shown in Figure 4-7.

9. Simulating Weapon Systems

Platforms in APS can be equipped with weapon systems by defining the weapon's characteristics, ammunition types, and sight reticle. A platform with a weapon system can engage and destroy any other platform, local or remote. Figure 4-8 shows the view through a TOW weapon sight looking at an attack helicopter. Weapon data structure definitions are contained in "weapons.h" which is reproduced in Appendix A. Current weapons include tank main gun with SABOT and HEAT rounds and the TOW antitank weapon system.

start_simtime() - Starts simulation time at 0.

stop_simtime() - Halts simulation time from advancing,
I.e. freezes time.

restart_simtime() - Restarts simtime when halted.

change_simspeed(float ratio) - Changes the ratio of
simulation time / real time. That is, a ratio value
of 0.3 will cause simulation to run 3 times slower
than real time, or 3 seconds of real time will elapse
for every 1 second of simulation time.

float read_simtimer() - Returns the current time since start_simtime
was called in simulation seconds.

void set_time_mark(void) - sets a time mark by storing current value
of simtimer in local static "package" variable.

float elapsed_time_wreset(void) - returns elapsed time in seconds
since time mark and resets time mark.

Figure 4 - 7 Simulation Timer Module Routines

Figure 4 - 8 View Through TOW Weapon Sight

Weapon system data structures are as general purpose as possible, to facilitate addition of other types of weapon systems and munitions. Each platform contains an array of pointers to onboard weapon systems, one of which is selected as the current weapon. Each weapon system is represented by an instance variable which contains data specific to that platform and pointers to class variables which contain generic data for that type of weapon or munition. The tank, for example, has a pointer to its weapon class variable and a pointer to the munition class variable for the particular type of round currently selected. Which sight reticle is drawn is determined by looking into the weapon class variable of the currently selected weapon for the currently selected platform. Presently tank main gun, binoculars, and TOW sight reticles are available[1].

Target ranging is also simulated for those weapons that normally have such a capability. The tank, for example, simulates a LASER range-finder by doing a **gselect** (similar to a graphic pick) on a three degree field-of-view along the firer's line-of-sight (LOS). The select list is examined for platform identifiers, except that of the firer, and returns the range to the closest one. This range is displayed in the weapon sight reticle. Platforms which no weapon system, such as jeeps, of course have no range-finding capability. They have, however, been provided with variable power binoculars selectable from the main driving menu.

Although range-finding with a LASER happens quickly enough so that it can be completed in a single iteration of the drawing loop, actions such as the flight of a round extend over multiple drawing cycles. In order to support such transient events without congesting every possible drawing loop in APS, an event handler which is called once each drawing loop was implemented. Events, such as a round in flight, are implemented as a linked list of event data structures (described in "weapons.h") with a common part containing a time stamp, delete flag, and pointer to

[1] Data is unclassified hypothetical data representing the generic characteristics of the represented item and is not intended to exactly match any actual system.

a function which can process this type of event. Each event also contains a variant part containing type-specific event data fields. Once each drawing loop the event handler is called. It traverses the event linked list. If an event is not marked for deletion, a call is made to its processing function via the pointer, with the address of the event record as the parameter. This allows the event processing function access to the type-specific data. Types of events currently implemented are:

1) **round_in_flight** - flies ballistic trajectory.

2) **reset_safety** - timeout to reset weapon safety after reload time.

3) **message** - displays message on screen for set period of time.

4) **splash** - draw splash of round miss for specified amount of time.

5) **flash** - draw expanding flash at impact of round with platform.

6) **bounce** - varies vehicle pitch based on elapsed time since going over bump in the terrain.

Firing a weapon in APS results in the simulation of the projectile's flight until impact with the ground, another platform, or maximum range is exceeded. The system assumes that the weapon system has some type of ballistic computer that will provide elevation to the weapon based on the range to the target, type of ammunition, etc.. This allows the position of the round along its ballistic path to be computed from a table of offsets in the Y (UP) direction, called its ballistic table, by scaling the offset using the current range from the point the round was fired. This produces an ordinate for the current range. A cylindrical viewing volume is then constructed along the LOS at the time the weapon was fired offset by the ordinate. LOS guided munitions such as the TOW are processed the same way, except that their ordinate is always zero and the flight volume is based on the firing platform's current LOS not the LOS at the time of firing. The near and far clipping planes are set to the starting and ending position of the round during the increment of time since the last update. This cylindrical volume is "swept" using **gselect** and the closest target to the firing vehicle is destroyed, if it is hit. If no target is in the volume, the round is checked for impact

with the ground and a splash drawing event is added to the event list. Finally, if the round flies beyond the edges of the terrain box, or exceeds its maximum range, it is terminated.

10. Module Descriptions

This section presents a brief description of the main modeling and simulation modules.

a. Program Control Flow

Program control flow is determined by state variables modified by the user through input from the dials, mouse, or menu system. Program structure is elaborated in Appendix A and the user interface including the menu system in Appendix C.

b. Supporting Routines

Supporting routines that perform a single function are too numerous to fully describe here. A listing of all modules is contained in Appendix A. Due to the incremental development of MPS, some modules overlap in function.

c. Data Structures

The data used to model vehicle motion is kept in the platform data structure defined in the "aps.h" (Appendix A). The platform data structure contains several state variables and toggles which are implemented as C enumerated types or a locally defined Boolean type. These state variables could be combined into a single variable using bit fields which would be more space efficient. This was not done due to the additional complexity of accessing bit fields and because bit fields are perhaps the *least* portable feature in ANSI C.

d. Turning/Steering Module

Steering is modeled using three routines:

- **float convert_course_to_turnrate(Vehicle *platform)** - Converts command course to turnrate which can be fed to the turning model. If the platform viewing mode is **driver**, the input is coming from the dials or the autopilot. This input is in the form of a commanded course or azimuth and the turnrate to direct the vehicle onto this course must be computed. This computed

turnrate is stored in the **cmd_turnrate** field of the platform record. This routine is implemented using the following rules:

1) If the difference between the command course and current course is less than a small delta, **CSE_WANDER**, then make them the same.
2) If the difference is less than **AUTO_TURNRATE**, then use difference as turnrate. Note that this may cause oversteer if the update time interval is greater than one second.
3) Otherwise use **AUTO_TURNRATE**.

- **update_platform_steering_model(Vehicle *platform, float elapsedsec, Boolean *network_packet_needed)** - This routine first calls **turning_model** to calculate the current turnrate. It then applies the current turnrate and time interval to calculate a new course. Finally, the viewing angles in the platform record are adjusted so that the view azimuth changes with the course.
- **float turning_model(float elapsedsec, float curr_turnrate, float cmd_turnrate)** - Returns exponential steering response if command turnrate is greater than current turnrate. If straightening out then centrifugal force is assisting so turnrate change is immediate.

e. Velocity Module

Consists of the routine **float velocity_model(float dt, float slope, float currvel, float cmdvel, float *pitch, Boolean *network_packet_needed)**. This routine returns the new platform speed using methods described in Chapter III. It also calculates and updates the transient vehicle pitch due to acceleration or braking. This transient pitch simulates the torque on the vehicle body during sudden velocity change as the vehicle body is constrained by the suspension system.

f. Bounce Module

Vehicle bounce due to changing terrain slope is started by routine **update_veh_pos** which sets the initial transient pitch angle amplitude if there is a change in vehicle pitch greater than **BOUNCE_THRESHOLD** (currently two degrees). Routine **handle_bounce** calculates a new transient pitch angle and updates the bounce amplitude field in the vehicle record. If the bounce amplitude has fallen below **PITCH_STEADY** then it is set equal to zero.

g. Math Module

Contains various general purpose math routines.

- **float convert_normal_to_slope(float normal[3])** - Returns the slope angle of a terrain polygon in radians based on surface normal.
- **transform_body_to_world(float azimuth, elevation, roll,**
  float dx, dy, dz,
  float *eye_x, *eye_y, *eye_z) - Transform *body* coordinates to *world* coordinates.
- **float calc_azimuth(float x1, float y1, float z1, float x2, float y2, float z2)** - Returns azimuth in radians from the positive X axis for a course from point1 to point2.

h. Path Operations Menu Module

This module (contained in **do_pathops.c**) contains the high level functions to create and delete a path, assign a platform to a path, and toggle the display of paths on and off. When called by selecting the "PATH OPERATIONS" option from the main driving menu, in module **do_driving_menu**, a popup menu is constructed and displayed. If a valid menu choice is made then the function selected is performed by calling one of the routines:

- **build_path** - Displays instructions, initializes a path structure, adds a point to the path and redraws the new path each time the left mouse button is pressed, prompts for a path name when right mouse button is pressed, adds path to path list, and updates path data file to add new path.
- **select_and_remove_path** - Displays instructions, when right mouse is pressed uses pick to determine which path was selected, deletes path from path list, saves remaining paths in path data file.
- **assign_veh_to_path** - Displays instructions, when right mouse is pressed uses pick to determine which vehicle icon was selected, makes cursor into vehicle icon, when icon cursor is moved over any point on a path and right mouse is pressed uses pick to determine which path is selected, makes copy of path for platform and sets platform's guide point to first point on the path. If platform is not local sends path over network to home simulator.

This module also contains functions to pick and display paths:

- **draw_path** - Draws a single path as a black line with a blue box around the first point and a red circle around the goal. Paths are drawn in overlay bit

planes so that it would not be necessary to redraw the entire 2D map each time a point is added to a path.

- **display_paths** - Displays all paths in normal drawing or pick mode depending upon its argument and returns the path identifier of the picked path.
- **pick_path** - Calls display_paths in pick mode and returns pointer to the path selected or NULL if no valid path is selected.

i. Path Module

This module (contained in "path.c") is a package which contains the low-level functions that operate on paths. The path data structure was shown in Figure 3-7. Paths are only manipulated using the functions in this module. Function prototypes are declared in "pathfunc.h". Since most of these functions operate on a specific path, most have a pointer to a PATH structure as one of the input arguments. Path points are kept as UTM coordinates and are converted to graphic system *world* coordinates as necessary. The following functions are supported:

- **addpt** - Adds a path point to the end of an existing path.
- **addpath** - Adds a path to the end of the path linked list.
- **at_goal** - Returns TRUE if path point has the same coordinates as the last point on a path.
- **copypath** - Copies path points from one path to another.
- **delete_path** - Deletes path structure and frees up space.
- **delete_list_path** - Deletes path from path list.
- **delete_veh_path** - Deletes platform copy of path.
- **init_path** - Returns pointer to a new path structure.
- **load_paths** - Loads paths from data file.
- **nextpt_on_path** - Returns pointer to the next point on a path.
- **reset_platform_path** - Clears platform path and reloads path originally assigned if any. Calls **start_down_path** to set initial guide point to path point nearest platform's current location.
- **save_paths** - Writes out all currently defined paths to binary file in the current default directory. The file structure description is contained in "pathdata.h".
- **set_guidept** - Replaces the platform's current guide point with input arguments and discards remaining path points.

- **start_down_path** - Returns pointer to path point closest to the input argument point (usually platform's current location).
- **update_guidept** - If platform is within VICINITY meters of current guide point and that guide point has a successor, set the platform's **guidept** field to point to next point on the path.

j. Autopilot Module

The autopilot works by setting the platform's commanded course and speed to follow its assigned path. For each local platform that has its **control** field set to AUTOPILOT and has a non-NULL **guidept** (i.e., it has a point to head towards) the autopilot performs the following functions:

1) Update the guide point if within a prescribed distance.

2) Handles obstacles (currently not implemented).

3) Update the platform's **cmdcse** to the azimuth from the platform's current location to its current guide point.

4) Sets commanded speed depending upon the current distance to the guide point. If the platform is so close that it might overshoot the guide point then braking is applied by setting **cmdvel** = -1.0. The platform's course is also frozen to avoid turning if the autopilot is engaged while the platform is near a guide point. Note that the current implementation does not control the platform with sufficient precision to navigate an obstacle field.

B. RULE-BASED PATH PLANNER

The path planner is implemented as three distinct levels. The top level is referred to as the path planner control program. It is through this program that overall control of the path planner is accomplished. The intermediate level consists of the search control program, which is implemented on a separate Symbolics workstation. The search control program controls access to the implemented search algorithm. Finally, at the lowest level is the implemented search algorithm. It is located on the same Symbolics workstation as its control program.

1. Path Planner Control Program

The path planner control program is located on the Symbolics workstation, SYM4. It is implemented using ART, a rule-based, expert system shell. The rules in this program control the action of all subordinate processes. There are 24 rules, grouped into the following seven categories.

- Set up
- Communications
- Clock actions
- Vehicle monitoring
- Vehicle and Path control
- Search control
- Fact clean up

This grouping of rules is used for conveyance of explanation, and does not necessarily have any bearing on the firing order of the rules. Appendix B contains a listing of the code.

a. Set Up Rules

The location of the vehicle simulation program and the search are variable as stated in Chapter III. This requires that the user input the location of these processes at the start up of the path planner control program. Two menu rules, **menu1** and **menu2**, are used for this. These in turn enable two communications start up rules, **start-iris-comm-links** and **start-sym-comm-links**. These communications rules open a TCP/IP I/O stream to the vehicle simulator process on the appropriate IRIS workstation, and a CHAOSNET I/O stream to the search control program on the appropriate Symbolics workstation. Program start up is the only time any of these set up rules are fired.

b. Communications Rules

The heart of the path planner control program's ability to monitor the actions of other processes on other machines is its ability to receive information. This

information is received via seven communications rules. Two of these rules, **check-comm-links-iris** and **check-comm-links-sym** are used to continuously check the I/O streams for incoming messages. These are the only communications rules in the path planner control program that are cyclic in nature. These rules are at the lowest active precedence level, and therefore do not cause any problem with indefinite postponement of other rules. The set up rules have a lower salience value but are only fired at program start up. These rules are cyclic because between the two of them they either assert facts that cause themselves to fire again, or cause rules to fire that in turn cause these two rules to fire again. These rules continue this cyclic action as long as there are no incoming messages.

When an incoming message arrives, one of the two previously mentioned rules asserts a fact indicating the type of message that arrived. Once this fact is asserted, one of four message handling rules reads in the message from the appropriate I/O stream and updates the knowledge base. These rules are: **read-init-in**, **read-update-in**, **read-map-ready-in**, and **read-waypoint-in**. The messages read in are as follows:

- Vehicle initialization message
- Vehicle update message
- Search map ready message
- Incoming waypoint message

c. Clock Rules

Each vehicle following a path computed by the path planner has a real time clock associated with it. The vehicle's clock is initially set to the time contained on the initialization message. This is done via the **set-clock** rule. When subsequent messages contain a time the vehicle's clock is reset to the message's time using the **reset-clock** rule. If no messages arrive over the networks the vehicle's clock is updated via the **update-clock** rule. This last rule enables the path planner control program to calculate a projected new position for the vehicle.

d. Vehicle Monitoring Rules

When a message arrives carrying vehicle update information, the **update-vehicle** rule modifies that vehicle's schema to reflect the new location, course, velocity, time and guidance mode. If no message arrives to update the vehicle's record, the **update-clock** rule gives the vehicle's **delta-time** fact a value. If the value of the vehicle's **delta-time** fact is positive, the **change-position** rule calculates the distance traveled, updates the vehicle's location, resets the vehicle's **delta-time** fact to 0, and indicates to the knowledge base that the vehicle has moved.

e. Vehicle and Waypoint Control Rules

When either the **update-vehicle** rule or the **change-position** rule fire, the knowledge base is updated to indicate that the vehicle has moved. This change to the vehicle's schema within the knowledge base fires the **check-for-new-waypoint** rule. This rule calculates the vehicle's current distance to the vehicle's current waypoint. If the distance to this waypoint is less than 200 meters, the vehicle's control schema is modified with the fact, (**new-waypoint yes**). When the knowledge database contains the control schema for a vehicle with the fact, (**new-waypoint yes**), the **send-new-waypoint** rule fires sending a new waypoint to the vehicle simulator. In this implementation, every other waypoint is skipped to mimic more closely the human commander's capability of skipping over 100 meter grid squares in his path planning.

f. Search Control Rules

After a vehicle has been initialized in the path planner control program's knowledge base, the **load-map** rule is fired. This rule tells the search control program to load a 10 KM by 10 KM map, with the specified lower left hand corner's UTM coordinates. After the map has been loaded and the search control program sends a message indicating that the search map is ready, the **start-path** rule fires. This rule gives the search control program the start point, goal point, and the vehicle's ID.

g. Fact Clean Up Rules

In order to prevent false firing of rules, used facts and schemas are cleaned out of the knowledge base whenever possible. The **clean-up-iris-msg** and **clean-up-sym-msg** rules clean up unclaimed waiting message facts. These facts are asserted by the **check-comm-links-iris** and the **check-comm-links-sym** rules when there is a message out of synchronization or spurious characters in the I/O stream. The **clean-up-waypoints** and **clean-up-vehicle** rules are fired when a vehicle goes out of guidance mode on the vehicle simulator. These rules remove all references to the vehicle from the path planner control program's knowledge base. This ensures that the next time the vehicle requests a path, an old path is not given.

2. Search Control Program

The search control program can be run from any Symbolics workstation, except SYM4 where the path planner control program resides. The search control program directly monitors the search algorithm and keeps track of which vehicles have maps and paths. The search control program receives two types of messages from the path planner control program. The first message requests that a 10 KM by 10 KM search map be loaded into a map array. This message specifies the vehicle and the UTM from the lower left hand corner of the 10 KM by 10 KM area to be searched. The second message requests that an optimal path be found from the start to the goal. This message specifies the vehicle, the start point's UTM, and the goal point's UTM. The map-array is a 102 by 102 grid. The size of this array is chosen to allow a search map with a resolution of 100 meter squares to be loaded into the map-array, including a border of non-traversable cells, to bound the search algorithm.

a. Loading the Map

When a message is received that requests a map be loaded, the search control program checks to see if that map has ever been loaded before. This is accomplished by first converting the map UTM and vehicle ID information, contained

as strings in the message, to LISP symbols and stored in **veh-map** and **current-veh**. The ***maps*** list is then checked to see if the newly generated map symbol is on the list. If the symbol is on the list, a message is sent to the path planner control program indicating that the map is loaded and ready to be searched. Alternatively the search control program builds a map-array with slope data from the data file used by the search algorithm. Finally, the symbol value of **veh-map** is then stored to the symbol value of the symbol stored in **current-veh**. The symbol value of **current-veh** is then added to the list ***vehs***.

b. Searching the Map

After the map is loaded, the search control program receives a message requesting a search be done for an optimal path. The message received contains the vehicle's ID, the desired path's start and goal points in UTM coordinates, as well as the map's lower left hand corner UTM coordinates. The UTM coordinates are converted to coordinates used by the wavefront search algorithm. The wavefront search algorithm is called with the appropriate map-array selected from the ***vehs*** list. The returned list of points is converted back to UTM coordinates. During this conversion a random number generator is used to move the waypoints around inside their 100 meter by 100 meter grid. This is done to simulate the commander's selection of a path from a low resolution map. Since the goal point may be a specific point, it is appended to the end of the list. Each waypoint is also tagged with the requesting vehicles ID and a sequence number. The sequence number is used by the path planner control program to keep track of waypoints.

c. Returning Waypoints

Waypoints are sent to the path planner control program one at a time for each path. This is accomplished through the **send-waypoints** function. The function is sent the list ***wave-paths***, every time the **search-controller** function cycles through its do loop. This list, that is sent to the **send-waypoints** function, contains, as separate lists, the remaining waypoints for every vehicle that has requested a path.

Each list is stripped of the first element, and this element, a waypoint, is sent to the path planner control program. This cycling continues until all of the waypoints have been transmitted.

D. SUMMARY

This chapter contains the description of the *implementation* of APS. The most salient modules and structures of the vehicle simulator are described with specific explanations of key code fragments and routines. The rules and control flow of the path planner is also described with a thumbnail sketch of each family of rules. This chapter explains how APS works while the following chapter describes the results of running the system.

V. SIMULATION RESULTS

APS achieves a large part of its research goals. A platform, depicted with a fair degree of realism, can be guided along a path, which is calculated in real-time, to its goal. However, direct comparisons of human and machine path planning are not possible due to a bottleneck in communications between the vehicle simulator and machine path planner. Also, due to time constraints, some capabilities were not implemented. The most important shortfall is local obstacles and obstacle avoidance.

A. VEHICLE SIMULATOR

The vehicle simulator achieves all the design capabilities listed in Chapter IV. Most importantly, it is able to support navigation of a platform along a designated path, under various combinations of manual and autonomous control. The path can be designated by a remote human commander or an AI machine. The remote commander can also turn the platform's autopilot and external guidance controls on and off, even while traversing a path under AI agent control.

The network communications supports connected multiple vehicle simulators with real time interaction supporting command and control and combat "dogfighting" capabilities. Simultaneous control of multiple platforms by different sources was demonstrated allowing local control of some platforms while others are controlled remotely by the AI agent.

The current vehicle simulator drawing cycle speed hovers near 4 frames a second. This speed produces jerky scene changes on the visual display and makes precise vehicle control difficult. However, it remains sufficiently realistic to support navigation over calculated paths. Run-time analysis indicates that steady state performance is bound by graphics operations and not computational load[1].

---

[1] CPU utilization was 50%-70% with the CPU waiting predominately for graphics calculations or drawing.

B. PATH PLANNER

Two key research goals of the path planner were to provide an easily understood interface between the vehicle simulation and the path planning algorithm, and to provide the mechanism for easy integration of search algorithms at the control interface level. The AI path planning program developed for this thesis has been tested in real time. The path planner supports the major goals of this thesis. It provides a functional interface whereby different search strategies can be evaluated and tested against a human planner. The AI path planner provided optimal paths for the driver of the simulated vehicle, using a wavefront search algorithm.

The path planner spends between one and one half to five minutes finding an optimal path. After the path is found the path planner control program begins returning waypoints. The issuance of waypoints along the path is not sufficiently fast enough to compete with the human planner. This is mainly due to the fact that the human planner issues an entire path to the vehicle at the beginning of the path, while the AI path planner is only allowed to issue one waypoint at a time. The AI path planner must apparently wait on buffered network communications. A "work around" exists to force the AI path planner to send the entire path as soon as the path is found. This however, would remove the path planner's ability to react to changes in the terrain as the vehicle travels along the path.

A rule-based path planner control program, written in ART, controls the flow of path requests and the issuance of waypoints. More than one simulated vehicle can be guided along a path at the same time. The path planner control program allows multiple vehicles to be guided as long as each has a unique vehicle ID.

The randomly generated offsets to the waypoints that are used to simulate a human path planner's waypoint selection within a one hundred meter square grid do not adequately simulate the way a human path planner selects waypoints along a path. The human planner generally chooses a path that transitions smoothly from grid to grid, except where demanded by terrain. The use of random numbers to select the

position of waypoints within the designated one hundred meter squares causes these waypoints to be unnaturally placed along the path. This can cause the simulated vehicle to make sharp changes in direction for no apparent reason.

The skipping of waypoints to provide a more reasonable next waypoint for the driver only appears natural in terrain that is typified by gradual changes in slope. Where the terrain changes slope frequently and drastically, the skipping of waypoints can cause the vehicle to traverse areas of extreme high cost. This occurs when the path planner has planned a route around a finger of a hill, but the waypoint avoiding the finger is skipped.

C. COMBINED SYSTEM

Obstacles, obstacle avoidance and local path planning were not implemented. Therefore, path transit time was purely a function of vehicle speed and an actual optimal path directly calculable from the global terrain data.

Several trials were run over identical routes (2-7 KM long) under human and AI agent path planning. Human path planning is relatively quick and accurate when there is distinctive terrain such as steep hills and flat valleys; that is, when the best route is fairly obvious. When terrain is mixed and the trade-off between going straight over steeper terrain or making a detour is more subtle, the visual decision becomes more difficult.

Since there were no obstacles, most trials were run with the autopilot. The autopilot always tries to maintain maximum speed, so a correctly calculated optimal path traversed on autopilot should result in a minimum transit time. Unfortunately, direct comparison between human and AI agent planned paths was not possible due to the inability of the Symbolics system to keep up with the vehicle simulator. Instead of the AI agent updating guide points when the vehicle was within 200 meters, so that there would be no break in speed, the vehicle would often reach a guide point and come to a stop before receiving the next guide point. When such a delayed guide

point was received, an additional time penalty was incurred as the vehicle simulator accelerated up to the maximum speed allowed by the terrain. As a result of this delay in receiving new guide points, transit times under the Symbolics AI agent control were 2-3 times longer than transit times for human planned paths. Consideration was given to working around this problem by having the AI agent send the entire path once calculated. However, this would eliminate the capability for the AI agent to dynamically modify the path, based on obstacles or other detours, so this option was rejected.

VI. SUMMARY AND CONCLUSIONS

A. LIMITATIONS

1. Vehicle Simulator

The APS vehicle simulator is currently limited in the following ways:

- Applies only to tactical vehicles travelling off-road.
- Models single gear transmission vehicle for acceleration.
- Operates in a single terrain database.
- Has simplified vehicle-terrain interaction model.
- Simulates joystick driving controls with a mouse.

There are some features of the vehicle simulator that don't work correctly or fail to work under certain special circumstances. A list of such features, the nature of the fault and all other known bugs is contained in Appendix D.

2. Path Planner

The AI path planner is currently limited in the following ways:

- The path planner does not take into account local obstacle avoidance.
- A vehicle can be run on an AI generated path only once.
- Only one Symbolics workstation is available to run the path planner control program written in ART.
- The terrain slope data file must be preprocessed into the correct format.
- The planned path is limited within a ten kilometer region.

B. AREAS FOR FURTHER STUDY

The most pressing need for further development is to remove the bottleneck at the AI agent end of the communications and to add obstacles and obstacle avoidance. Breaking the path planner's message processing logjam would allow direct comparisons between the actual transit time of human and machine planned paths, a major goal of this research. Obstacles would add the global-local dimension to functional assignment trade-offs between human and machine planners, another

unexplored area. Other areas for further study lie in increased realism and added functionality for the vehicle simulator with multiple algorithms the focus for the path planner.

3. Vehicle Simulator

The most fruitful areas for further study of the vehicle simulator are simulator realism, graphics performance, local autonomous operations, and program structure/software engineering issues.

a. Program Structure / Software Engineering

The vehicle Simulator program consists of ≈37,000 lines of source code in 238 files. About 7500 lines are pure drawing code, that is, polygons and figures. The majority of the source files contain a single function. This flat program structure in such a large program doesn't provide the modularity or encapsulation necessary to manage the rapid modification and maintenance necessary in a system subject to the constant flux of research. For example, to add a new platform type, say an Armored Personnel Carrier (APC), would necessitate modifying more than 20 files, even if its graphical object definition, material definitions, and vehicle characteristics were already available. A requirement to modify the platform modeling or control for this new vehicle type would entail even more extensive and treacherous changes.

An Object Oriented Programming (OOP) Language would provide an order of magnitude simplification of the program structure and code. Encapsulation would limit the effects of code modifications reducing debugging and retesting of working components ensure the containment of side effects. Inheritance would eliminate duplicating code that performs essentially the same thing but in slightly varying ways. For example, this would allow each platform to be an instantiation of a general class containing methods for control, modeling, and display. These methods would then operate on class and local data structures to provide the required function.

Since the vehicle simulator is written in C and currently C seems to have the most thorough and efficient interface to the SGI graphics library, a C based OOP language such as C++ or ObjectiveC would be appropriate candidates for such a conversion, with a low risk that performance penalties might eliminate its advantages.

Another alternative is Ada. Encapsulation and inheritance can also be implemented through Ada "packages". In addition Ada is expressive enough to serve as a program design language (PDL) and is, after all, the DoD "standard" language.

b. Realism

Current research at NPS has produced some capabilities that could enhance realism without a large performance penalty. The realism of the 3D depiction of terrain can be improved by increasing the resolution of the terrain data. This shrinks the size of the near view terrain polygons making them seem more natural. In addition, the terrain display could be made more realistic by adding "features" such as roads, structures, lakes, and vegetation. Winn and Strong [WINN&89] have demonstrated a terrain drawing system that, utilizing IRIS hardware support, increases the terrain resolution, helps realism through better shading techniques ***and*** boosts performance. They also developed a real-time line-of-sight system that could be useful as a alternative or additional cost function for path planning. Adoption of a standard graphical object definition language such as Pixar's Renderman or Object File Format (OFF), the language developed at NPS [MUNSON89] would create access to a large library of realistic images of platforms and other objects.

Vehicle realism could be enhanced by including the following features:

- Multiple gear transmissions.
- Realistic slope effects.
- Sound.
- Different model constants by vehicle type.
- Adding vehicle stability effects; i.e., turn over or crash.
- Energy/Fuel consumption.

c. *Increased Capability*

APS is currently limited to preprocessed terrain data for one 35 square kilometer area of the world. Drummond and Nizolak [NIZOLK&89] in FOST modified the original MPS terrain representation system to accept standard format DTED files, available for many parts of the world.

Additional path planning cost functions, such as exposure to enemy observation, energy or fuel consumption, tactical maneuver advantage, etc., could be used as alternate or combined figures of merit to evaluate the quality of the product of the path planner in differing environments.

The trafficability model could be expanded from a simple function of slope magnitude to consider soil conditions, vehicle traction, and anisotropic slope effects such as those contained in the vehicle-terrain interaction model of Ross [ROSS89].

The unimplemented simulated vision system, planned to provide input on local conditions to the obstacle avoidance system, was modeled after the laser terrain scanning system of the Adaptive Suspension Vehicle [BIHARI&89: pg 61]. There is an interesting interaction between the range and resolution of the vision system, the speed of the obstacle avoidance process and the maximum safe speed of the vehicle. Simply put, the vehicle cannot safely go faster than its sensing and navigation system can react and respond. Were local obstacles and obstacle avoidance implemented, this simulator could be used to compare the overall performance of vision systems by varying the sensing system parameters: range, resolution, field-of-view, and speed; and then navigating *real* terrain.

d. *Performance*

Vehicle performance in terms of frames per second is of concern in the vehicle simulator only insofar as it effects realism. Other researchers at NPS have looked specifically at performance and found no magic algorithm that promises orders of magnitude improvement due to software changes [FICHTN&88]. That does not

mean that performance comparisons are unimportant or that efficiency can be ignored, simply that performance is not *directly* germane to this research.

4. Path Planner

Two key research goals of the path planner used in this thesis were first, to provide an easily understood interface between the vehicle simulation and the path planning algorithm, and to provide the mechanism whereby search algorithms could be easily interchanged at the control interface level. This implementation of the path planner is a prototype that needs to be refined and expanded. Areas of research that would provide significant improvements on this study are as follows:

- Incremental route planning
- Selection of route planning algorithms depending on requirements
- Comparisons of expert system shells
- Comparisons of search algorithms using real terrain data and simulated vehicles
- Improved communications

a. *Search Algorithms*

The wavefront search algorithm used in this study is well understood and provides a standard by which other search algorithms can be judged. There are many other algorithms available that provide capabilities unique to each. The decision to use a particular search algorithm may be based on the constraints of the path and mission. An area for further study is to select appropriate search algorithms, dependent on the terrain and mission to be planned. Another area of study is the use of a preprocessing algorithm that would allow the vehicle to start along the path before the path is completed and still get reasonable results.

b. *Expert System Shells*

The path planner was implemented in ART which provides a high level symbolic programming environment that allows predicate representation of rules. This representation allows the path planner written in ART to be understood by any-

one who has a grasp of predicate logic. ART however is not currently supported on the next generation of Symbolics workstations, nor is ART code easily converted to some common language and then transported to some other LISP machine. This last area of research is particularly interesting as graphics machines are beginning to incorporate LISP processors as an integral part of the architecture.

c. *Communications*

The path planner has not been able to keep up with the vehicle simulation. There appears to be a problem with the buffering of messages in the Symbolics workstations. The path planner could also be improved by the addition of algorithms that would check for the most recent update message instead of filtering down through the messages that have arrived and backed up in the buffer.

C. SUMMARY

APS provides a testbed for the study of real-time path planning and control strategies and algorithms without the cost of building actual hardware. It serves as a bridge between the theoretical study of a simplified abstract problem to applied research producing concrete performance under realistic conditions. The conclusions of this study show the feasibility and advantages of such as system in settling performance debates with empirical results.

APPENDIX A. VEHICLE SIMULATOR MODULE DESCRIPTIONS

A. DATA DESCRIPTIONS

Data structure *definitions* and program constants are contained in C "header files" which normally have an "h" suffix. A list of all APS header files is contained in Table A-1. The main data structures used in APS are contained in "aps.h" (Figure A-1) and "weapons.h" (Figure A-2). Global variable *declarations* are contained in "global.h" which is *included* at compile time in the APS main module "aps.c".

B. MODULE ORGANIZATION AND PROGRAM CONTROL FLOW

The top level APS function **main()** (Figure A-3) is contained in the file "aps.c". This module initializes the system, runs the simulator by calling **event()**, and cleans up during program termination. Module **event()** (Figure A-4) initializes the simulator, displays introductory screens, gets a user selection of a 10 Kilometer area to work in, handles the main menu selections and calls either of the two main drawing loops: **event_driving()** (Figure A-5) or **event_flying()** (Figure A-6). If the user selects RETURN TO MAIN MENU from the driving menu or the platform he is operating is destroyed, control returns to **event()**, the 10 Kilometer 2D map is displayed, and the main menu is presented to renew the cycle. If the user selects EXIT THE PROGRAM from the driving menu control then the program is terminated by returning control through **event()** to **main()**. The remaining modules contain functions which are either sub-packages under one of the main routines or general support functions that are called to do some task from several places. These modules, the functions contained in each one, and a brief description of what they do are listed in Table A-2.

TABLE A-1 APS HEADER FILES

aps.h	Main global data structures and constants.
Cobra_data.h	Cobra object data.
Cobra_inside_pt.h	Object data for Cobra inside view.
color_scheme.h	Program RGB color array indexes.
controls.h	Constants for controls.
event_status.h	Main loop state definitions
files.h	System data file names
flamedata.h	Object data for wreck (burning jeep flames).
global.h	Global variable declarations.
gundata.h	Tank main gun object data.
jeepdata.h	Jeep object data.
legend.h	Positioning constants for legend windows
lightcons.h	Material definition constants.
lightdefs.h	Lighting array declarations.
macros.h	Copy of system header file that defines C macros without bug.
Main_rotor_data.h	Object data for Cobra main rotor.
math_utility.h	Math_utility function prototypes.
missiledata.h	FOGM object data.
Mrotor_inside_pt.h	Object data for tip of main rotor seen from inside Cobra cockpit.
network.h	Network message delimiters, types and formats.

TABLE A-1 APS HEADER FILES - CONTINUED

network_services.h	Network function prototypes.
openjeepdata.h	Open jeep object data.
pathdata.h	Path data structures definitions.
pathfunc.h	Path function prototypes.
popups.h	Popup menu names and return values.
rollerdata.h	Tank roadwheel object data.
Rotdat.h	Cobra rotor rotation rates.
Tail_pipe_data.h	Cobra IR suppressor object data.
Tail_rotor_data.h	Cobra tail rotor object data.
tankdata.h	Tank object data.
terrain.h	Terrain 3D display constants.
tiredata.h	Wheeled vehicle tire object data.
Tpipe_inside_pt.h	Cobra IR suppressor object data.
trackdata.h	Tank track object data.
trackdata2.h	More tank track object data.
Trotor_inside_pt.h	Cobra tail rotor data.
truckdata.h	Truck object data.
turitdata.h	Tank turret object data.
vehmodel.h	Platform motion modeling constants.
weapons.h	Weapons system data structures and constants.

```
#include "gl.h"
#include "fmclient.h"  /* inherit font manager definitions */
#include "pathdata.h"
/* pathdata.h required because there are ptrs to paths in Vehicle
   data structure.
*/

#ifndef  NULL
#define  NULL              0
#endif


/* defines for manipulating the terrain data file */
#define  ELEV_MASK         0x1fff
#define  RD                0
#define  WR                1

/* defines for polygon computations */
#define X  0   /* X coordinate */
#define Y  1   /* Y            */
#define Z  2   /* Z            */
#define L  0   /* LOWER triangle */
#define U  1   /* UPPER triangle */

/* defines for polygon orientation */
#define MAXCOORDS 80

#define  MAXLOOKDISTF      32808.0

/* define maximum size for pickbuffer */
#define  PICK_BUFFER_SIZE  512

/* define default range for rangefinder */
#define RANGE_DEFAULT  9999
```

Figure A-1 APS.H Main Header File

```
/* defines for conversions */
#define  TO_MPS            0.447039
#define  FEET_TO_METERS    0.3048

/* defines for useful constants */
#define  TENTHKM           100.0
#define  TWOTENTHKM            200.0
#define  HALFKM            500.0

#define  ONEKM             1000.0
#define  TWOKM             2000.0
#define  TENKM             10000.0
#define  MAXLOOKDIST       5943.6  /* 5943.6 meters = 19500 feet */
#define  MAXELEV           1134    /* 1134 meters = 3720 feet */
#define  MINELEV           0       /* 0 meters = 0 feet */
#define  NUMXGRIDS         100
#define  NUMZGRIDS         100
#define  X_DATA_PTS        101
#define  Z_DATA_PTS        101
#define  TANKGNDHT         1.6612
#define  TRUCKGNDHT        1.6764
#define  JEEPGNDHT         0.6857
#define  OPENJEEPGNDHT     0.6857
#define  TOWGNDHT          1.0
#define  ATKHELHT          1.0

#define  MAXVEH            999
         /* Maximum number of LOCAL platforms allowed */
#define  MAXVEH_NUMBITS    10
         /* Number of bits to shift host id to make room for MAXVEH
ids */
#define  VEHID_MASK        0xFFFFF
         /* Mask to get positive local base platform id */

#define  MAXDEFAULTS       9
#define  FOGM_INIT_HT      50.0
```

Figure A-1 APS.H Main Header File - Continued

```
/* defines for miscellaneous trig operations */
#define  QTR_PI           0.785398163
#define  HALFPI           1.570796327
#define  THREE_QTR_PI     2.35619449
#define  PI               3.141592654
#define  FIVE_QTR_PI      3.926990817
#define  THREE_HALVES_PI  4.71238898
#define  SEVEN_QTR_PI     5.4977871
#define  TWOPI            6.283185307
#define  RTOD             57.29577951
#define  RTOD_X_10        572.9577951
#define  DTOR             0.017453292

/* defines for cursor related stuff */
#define  ARROW            0
#define  TANKCURSOR       1
#define  TRUCKCURSOR      2
#define  JEEPCURSOR       3
#define  FOGMCURSOR       4
#define  WRECKCURSOR      5
#define  OPENJEEPCURSOR   6
#define  COBRACURSOR      7
#define  CROSSHAIR        8
#define  XCURSOR          9
#define  BOXCURSOR        10
#define  BLANKCURSOR      11
#define  HRGLASSCURSOR    12
#define  STEERCURSOR      13

/* platform types */
#define  NUMVEHTYPES      8
#define  TANK             0
#define  TRUCK            1
#define  JEEP             2
#define  FOGM             3
#define  WRECK            4
#define  OPENJEEP         5
```

Figure A-1 APS.H Main Header File - Continued

```
#define  TOWVEH          6
#define  ATKHEL          7

/* upper limit on weapon types per platform */
#define MAXWEAPONS              2
#define MAX_RDTYPES_PER_WEAPON  2

/* defines for the arrows drawn on the 2D terrain map */
#define  ARROW_LENGTH          30.0
#define  ARROW_WING_LENGTH     10.0
#define  ARROW_WING_ANGLE      25.0

/* defines for window ids */
#define  BILLBOARDWIN     0
#define  MAPWIN           1
#define  MENUWIN          2
#define  NAVWIN           3
#define  INDWIN           4

/* Overdraw color defines */

#define CLEAROVERDRAW     0
#define BLACKOVERDRAW     1
#define REDOVERDRAW       2
#define BLUEOVERDRAW      3

/*  defines for networking */
#define PACKET_SIZE    512

            /******   Data type definitions     *******/
typedef float  time_f;  /* time, in floating point seconds */

/*  enumerated variable to indicate which viewing mode driven vehicle
is in */
typedef enum     {  normal_view = 0,
                    driver,
                    binoculars,
                    wpn_sight   } View_modes ;
```

Figure A-1 APS.H Main Header File - Continued

```
/*  Define enumerated type for global variable to indicate current
    platform control mode.
*/
typedef enum { MANUAL = 0,
               AUTOPILOT } Control_type;


typedef  enum  {  LOCAL = 0, NET } Vehowner;  /* Origin of platform,
                                                 local or network.  */

typedef enum  {   OFF = 0, ON  } Toggle;  /* Indicates whether
                                             something is on or off.  */
/* type definitions for platform data structure */
typedef struct vehicle {
int   net_id;       /* PLATFORM ID NUMBER FOR NETWORKING PURPOSES*/
short pick_id;      /* PICK ID NUMBER FOR TARGETING PURPOSES */
Vehowner owner;     /* indicates whether platform is local or net */
Control_type control;  /* Indicates how this platform is controlled.*/

View_modes  view_mode; /* Indicates desired view from vehicle */
Toggle      ext_guidance;  /* Indicates whether external guidance
                              is ON or OFF.  */
Toggle      recv_path;     /* Indicates whether incoming guidepts
                              should be added onto existing path */
Boolean     send_update;   /* Flag indicating whether update should
                              be sent out over network for this
                              platform.
                           */
short       t;      /* PLATFORM TYPE */
Coord       x;      /* X TRANSLATION */
Coord       y;      /* Y TRANSLATION */
Coord       z;      /* Z TRANSLATION */
double  utm_x,
        utm_y;     /* UTM Coordinates of platform to meter */
float    cse;      /* veh heading, (rotation about Y axis)
                      from positive X-axis, in radians. Must
                      be converted to compass degrees for I/O. */
```

Figure A-1 APS.H Main Header File - Continued

```
float cmdcse;     /* Desired ( directed ) vehicle course */

float   turnrate;     /*  psi dot - current turning rate */
float  cmd_turnrate; /*  desired psi dot - either manually input
                         through driving controls or calculated
                         when change of course received from
                         remote controller.
                      */
float   base_pitch;  /* veh pitch (tilt) around Z axis, due to
                        slope.
                      */
float   trans_pitch; /* transient vehicle pitch offset due to
                        acceleration, vehicle bounce, etc.
                      */
float   base_roll;   /* veh roll angle, around X axis (cant), due
                        to slope.
                      */
float   trans_roll;  /* transient vehicle roll induced by centrifugal
                        force when turning.
                      *
float bounce_amplitude; /* amplitude of platorm pitch oscillations */
float bounce_time;      /* accumulated time in seconds since bounce
                           started. */
float   wpnaz;      /* weapon system azimuth, radians from X axis */
float wpnelev;      /* weapon system elevation from horizontal */
float  viewaz;      /* viewer's angle of view from X axis in radians */
float viewelev;     /* viewer's elevation angle (tilt) */
float     vel;      /* VELOCITY IN METERS PER SECOND */
float  cmdvel;      /* Platform control setting  */
                    /* Represents the velocity commanded for the
                       platform either by an outside controller or
                       by setting the throttle control at a certain
                       setting.  A positive difference cmdvel - vel
                       means acceleration and a negative difference
                       means coasting to new lower velocity.
                       A negative value for cmdvel is interpreted
```

Figure A-1 APS.H Main Header File - Continued

```
                    as a braking factor, range -1.0 <= v < 0.0.
                  */
float     alt;     /* ALTITUDE IF IT IS A FOG-M MISSLE */
Boolean track_flag; /* IF TYPE IS A GROUND PLATFORM THEN */
                    /*    FALSE = NOT BEING TRACKED    */
                    /*    TRUE  = IS BEING TRACKED     */
                    /* IF TYPE IS A FOGM MISSILE THEN  */
                    /*    FALSE = NOT CURRENTLY TRACKING */
                    /*    TRUE  = IS TRACKING          */
struct vehicle *track;  /* IF TYPE IS A GROUND PLATFORM THEN */
                        /* IT IS A POINTER TO THE FOGM, OTHERWISE */
                        /* IT POINTS TO THE GROUND PLATFORM */
/*  WARNING: Following is ANSI C "incomplete structure definition" of
    structure contained in weapons.h.  Such forward declarations MAY
    not be supported in non-ANSI C compilers.
*/
struct weapon_record *wpnptr[MAXWEAPONS]; /* AVAILABLE WEAPONS */
struct weapon_record *wpn_selected;       /* CURRENT WEAPON */
/* Fields containing pointer to associated path and the current
   guide point being used to navigate vehicle.
*/
PATH                *path;
PTNODE              *guidept;

struct vehicle *next;   /* NEXT NODE IN THE LIST */
      }  Vehicle

/* type definition for fired weapon event */
typedef struct  {
            int         firer_id;
            Coord       fired_x, fired_y, fired_z;
            Coord       tgt_x, tgt_y, tgt_z;
            float       wpnaz, wpnelev;
            } FIRE_EVENTS;

/* declare extern functions (alphabetical order ) */
extern  float       arcsine();
```

Figure A-1 APS.H Main Header File - Continued

```
extern  float       calc_distance( Coord x1, Coord y1, Coord z1,
                                   Coord x2, Coord y2, Coord z2 );
extern  void        center_string_map( char *str, long linenum );
extern  void        center_string_menu( char *str, long linenum );
extern  float       compass_degrees_to_radian_angle( float deg );
extern  float       convert_to_dec_hr();
extern  short       convert_to_hr_min();
extern  time_f      elapsed_time_wreset(void);
extern  Vehicle     *find_platform( int netid );  /* in
check_for_packets.c */
extern  float       gnd_level( Coord x, Coord z );
extern  mousescreentoutm( short sx, short sy,
                          double *utmx, double *utmy,
                          short mousew);
extern  mouseutmtoscreen( double utmx, double utmy,
                          short *sx, short *sy,
                          short window);
extern  mouseutmtoterrain( double utmx, double utmy, float *tx, float
*tz);
extern  mouseterraintoutm( float tx, float tz, double *utmx, double
*utmy);
extern  float       radian_angle_to_compass_degrees( float angle );
extern  time_f      read_simtimer();
extern  float       restrict_angle_to_first_revolution( float angle );
                                    /* radians only */
extern  float       sincos( float angle, float *cosine );
extern  Vehicle    *switch_veh();
extern  short       tot_num_ground_veh();
extern  short       tot_num_veh();
extern  double      vecdotp();
extern  double      vecmag()


/* declare very common global variables */
extern  Vehicle *vehlist, *vehlistend, *driven;
extern  Boolean networking;
extern  int     color_scheme_index;
```

Figure A-1 APS.H Main Header File - Continued

```
extern  int     guidance_signal;
extern  float   eye_position[NUMVEHTYPES][3];
                /* offset of eye from center of veh */
extern  long  centerx, centery; /* screen coord of center of MAPWIN */
extern  Boolean control_connected;/* flag indicating remote process is
                                     connected to server.
                                  */

/* Font handles  -  Initialized in initiris */
extern  fmfonthandle   Helv, HelvB, TimesRm, TimesRmB;
extern  fmfonthandle   scaled_TimesRm, scaled_TimesRmB,
                       scaled_Helv, scaled_HelvB;
/*  Make UTM coordinates of lower left corner of 10 KM box global */
extern   double   LL_tenkmutm_x, LL_tenkmutm_y,
                  UR_tenkmutm_x, UR_tenkmutm_y;
/* Coord of Lower left of zoomed box */
extern   double   zoomed_LL_x, zoomed_LL_y;
```

Figure A-1 APS.H Main Header File - Continued

```
/*********************************************************************;
NAME      : weapons.h
CALLED BY :
CALLS     :
MODIFIED  : 12/14/88
PERSON    : Bill Teter
PURPOSE   : Contains record and type definitions for weapons,
 ammunition types, and sight reticles for weapons systems.  Uses some
types from MPS.H so it must follow it in include statements.

*********************************************************************/

/****  Sight Types  ****/
#define  NORMAL      0
#define  M1TANK_MG   1
```

Figure A-2 WEAPONS Header File

```
#define  BINOS        2
#define  TOW          3

#define  DRAGON       4
#define  IFV_SBT      5
#define  IFV_HEI      6

#define  IFV_TOW      7
#define  COBRA_TOW    8
#define  COBRA_20MM   9
#define  APACHE_HF    10
#define  APACHE_25    11

#define  TANKFIRE_INTERVAL  6.0
         /* Reload time for generic tank system */
/* Define FOV and ASPECT to set perspective when detecting hit by
   picking.  */
#define  ROUND_FOV        3
         /*  .3 degrees or 6 mils */
#define  ROUND_ASPECT     0.8
#define  SPLASH_DURATION 4.0
          /* How long to display target miss ground splash */
#define  FIRING_TBL_INC 100.0
         /* Range increments in firing tables */
#define  FIRING_TBL_LENGTH 10
         /* Number of entries in ballistic tables */;

typedef struct worldcoord_2D {   float x,y; } WCOORD2;
typedef struct worldcoord_3D {   float x,y,z; } WCOORD3;
typedef struct screencoord   {   short x,y; } VCOORD;
typedef short Colorvector_S[3];
typedef long  Colorvector_L[3];
typedef float Colorvector_F[3];
```

Figure A-2 WEAPONS Header File - Continued

```
/*-----------------------------------------------------------------
--    Define record for class of ammunition or type  of round      --
-----------------------------------------------------------------

     +-----------------------------------------------------+
     |                trajectory_type ( ENUM )             |
     +-----------------------------------------------------+
     |                warhead  ( ENUM )                    |
     +-----------------------------------------------------+
     |                round_name ( STRING[10] )            |
     +-----------------------------------------------------+
     |                speed  ( meters/sec )                |
     +-----------------------------------------------------+
     |                minrange  ( meters )                 |
     +-----------------------------------------------------+
     |                maxrange  ( meters )                 |
     +-----------------------------------------------------+

*/
typedef enum   gdtype { BALLISTIC, GUIDED_LOS } TYPE_FLIGHT ;
typedef enum   ammo_type { INERT, CHEMICAL, NUKE, FASCAM,
                           SUBMUNITION } TYPE_WARHEAD ;
typedef  struct munition_type {

      TYPE_FLIGHT    trajectory_type;   /* type of trajectory */
      TYPE_WARHEAD   warhead;           /* type of warhead    */
      char           round_name[10];    /* string name of munition */
      float          speed;             /* meters/second */
      float          minrange,          /* minimum arming range */
                     maxrange;          /* maximum effective range */
      float          ballistic_table[11];

      } MUNITION_CLASS;;

/*-----------------------------------------------------------------
--             Define structure for sight reticle               --
-----------------------------------------------------------------
```

Figure A-2 WEAPONS Header File - Continued

```
      +-----------------------------------------------------+
      |                        type                         |
      +-----------------------------------------------------+
      |                   name - STRING                     |
      +-----------------------------------------------------+
      |                   magnification                     |
      +-----------------------------------------------------+
      |               numlines - # lines in reticle         |
      +-----------------------------------------------------+
      |               lines  --> HAIRLINE (array)           |
      +-----------------------------------------------------+
      |                   safe_light                        |
      +-----------------------------------------------------+
      |          range_posn ( posn of range string )        |
      +-----------------------------------------------------+
      |          round_posn ( posn of round string )        |
      +-----------------------------------------------------+
*/
typedef  int   CHAR_POSN[2];  /* X,Y tuple defines origin of text */
/* Lower left and upper right, dimensions of a rectangle */
typedef  struct rect_type_2D  {
         float    ll_x, ll_y, ur_x, ur_y;
         } RECTANGLE2D;
typedef  struct hairline_record { /* reticle hairline start - end */
         float    start[2], end[2];
         } HAIRLINE;;

typedef  struct reticle_record  {

int   type;             /* code for type of reticle  */
char  name[10];         /* string containing weapon/reticle name  */
short magnification;    /* normal magnification of sight */
int   numlines;         /* count of number of hairlines in reticle */
HAIRLINE *lines;        /* pointer to any array of HAIRLINE        */
RECTANGLE2D safe_light; /* rectangle coordinates for safety light */
CHAR_POSN   range_posn; /* origin of range string */
CHAR_POSN   round_posn; /* origin of round name string */
   } RETICLE;
```

Figure A-2 WEAPONS Header File - Continued

```
/*-------------------------------------------------------------------
--  Define structure to represent fired round while in flight      --
-------------------------------------------------------------------*/

typedef  struct RIF_types {
   WCOORD3          fire_posn,      /* location of platform when
                                       round was fired. */
                    posn,           /* location at last update */
                    pt_of_aim;      /* world coord pt of aim */
   float            fired_range,    /* range data used when fired */
                    fired_dist;     /* current distance from pt where
                                       fired */
   float            angle,
                    elev;
  Vehicle          *firer;         /* ptr to platform that fired
                                      the round*/
   MUNITION_CLASS   *ammo;         /* type of round  */
} ROUND_IN_FLIGHT;;


/*-------------------------------------------------------------------
--  Define structure for class of weapon systems.  There will be   --
--    one of these records for each type of weapon system.         --
-------------------------------------------------------------------

     +-----------------------------------------------------+
     |                    name - STRING                    |
     +-----------------------------------------------------+
     |                sight  --> RETICLE                   |
     +-----------------------------------------------------+
     |                reload_time  ( time )                |
     +-----------------------------------------------------+
     |     ammo_types ( array of --> MUNITION_CLASS )      |
     +-----------------------------------------------------+
     |                basic_load ( array of int )          |
     +-----------------------------------------------------+
*/
```

Figure A-2 WEAPONS Header File - Continued

```
typedef  struct   weapon_type {

   char     name[20];     /* name of weapon sys, ex: M1Tank MG */
   RETICLE  *sight;       /* sight picture used for this weapon type */
   time_f   reload_time; /* Minimum time between firings  */

   /*  Array of pointers to posible munitions for this weapon sys */
   MUNITION_CLASS    *ammo_types[MAX_RDTYPES_PER_WEAPON];
   /*  Array holding starting quantities for avail munitions */

   int               basic_load[MAX_RDTYPES_PER_WEAPON];

} WEAPONS;

/*-------------------------------------------------------------------
--  Define structure to represent weapon system carried by a       --
--  platform.                                                      --
-------------------------------------------------------------------
     +-----------------------------------------------------+
     |                wpn_class --> WEAPONS                |
     +-----------------------------------------------------+
     |                range_reading                        |
     +-----------------------------------------------------+
     |                safety_on  ( refire FLAG )           |
     +-----------------------------------------------------+
     |            round_select --> MUNITION_CLASS          |
     +-----------------------------------------------------+
     |                last_fired  ( time )                 |
     +-----------------------------------------------------+
     |      rounds_remaining[] ( array by type round )      |
     +-----------------------------------------------------+
*/

typedef  struct   weapon_record  {  /* instance of weapon */

/* class variable */
WEAPONS  *wpn_class;    /* ptr to weapon type record */
```

Figure A-2 WEAPONS Header File - Continued

```
   /* instance variables */
float    range_reading; /* current reading in rangefinder */
Boolean  safety_on;     /* flag whether weapon safety is on or off */
/* ptr to selected round, type of munition currently selected   */
MUNITION_CLASS     *round_select;
/*  time system was last fired, 0 if never fired. */
time_f               last_fired;
/*  array of rounds of each type of munition remaining on platform */
int      rounds_remaining[MAX_RDTYPES_PER_WEAPON];

   } WEAPON;

/*-----------------------------------------------------------------
--   Define record structure for timed events                    --
-----------------------------------------------------------------

     +-----------------------------------------------------+
     |                delete   ( FLAG )                    |
     +-----------------------------------------------------+
     |                start_time   ( time )                |
     +-----------------------------------------------------+
     |                last_update ( time )                 |
     +-----------------------------------------------------+
     |              process_event --> func( -->event )     |
     +-----------------------------------------------------+
     |                next_event   --> event               |
     +-----------------------------------------------------+
     |                variant   ( UNION )                  |
     +-----------------------------------------------------+

*/;

/*  Record for event variant part to reset something on a weapon after
    a certain amount of elapsed time. */
typedef struct weapon_timeout {

     time_f              duration;
     WEAPON              *wpnptr;
```

Figure A-2 WEAPONS Header File - Continued

```
     } WPN_TIMEOUT;


typedef struct msg_type {

     time_f              duration;
     char                message[40];

     } MESSAGE_TYPE;


typedef struct splash_record {

     Coord               x,y,z;     /* Where to draw round splash */

     } SPLASH_EVENT;


typedef struct flash_record {

     short               colornum;
     Vehicle             *hitvehicle;

     } FLASH_EVENT;

typedef struct {    /* Record for vehicle "bounce" */

     Vehicle  *vehptr;
     float    bounce_amplitude;

     } BOUNCE_EVENT;

typedef union   type_events   {

      ROUND_IN_FLIGHT    round_aloft;
      WPN_TIMEOUT        wpntimeout;
      MESSAGE_TYPE       letter;
      SPLASH_EVENT       splash;
```

Figure A-2 WEAPONS Header File - Continued

```
      FLASH_EVENT         flash;
      BOUNCE_EVENT        bounce;
      /*  add other timed event types here */
} EVENT_UNION; /* end union */

typedef  struct   event_record {
      Boolean     delete;          /* Flag indicating expired event */
      time_f      start_time,      /* time event was initiated */
                  last_update;     /* time when event was last
                                      updated  */
      int         (* process_event)(struct event_record *);
                  /* pointer to function to handle this event */
      struct event_record *next_event;
      EVENT_UNION         variant;  /* variant part of record */
} EVENTS;  /* end struct event_record */

/*-------------------------------------------------------------------
--  Declare global variables to contain the values for actual      --
--  WEAPONS, RETICLE, and MUNITION_CLASS classe                    --
-------------------------------------------------------------------*/

extern    WEAPONS   m1_tankmg_sabot,
                    m1_tankmg_heat,
                    binos_class,
                    tow_class;

extern    RETICLE   m1tank_gunner_reticle,
                    binos_reticle,
                    tow_reticle;

extern    MUNITION_CLASS      m1_105sabot,
                              m1_105heat,
                              tow_standard;

extern    WEAPON    binos;
```

Figure A-2 WEAPONS Header File - Continued

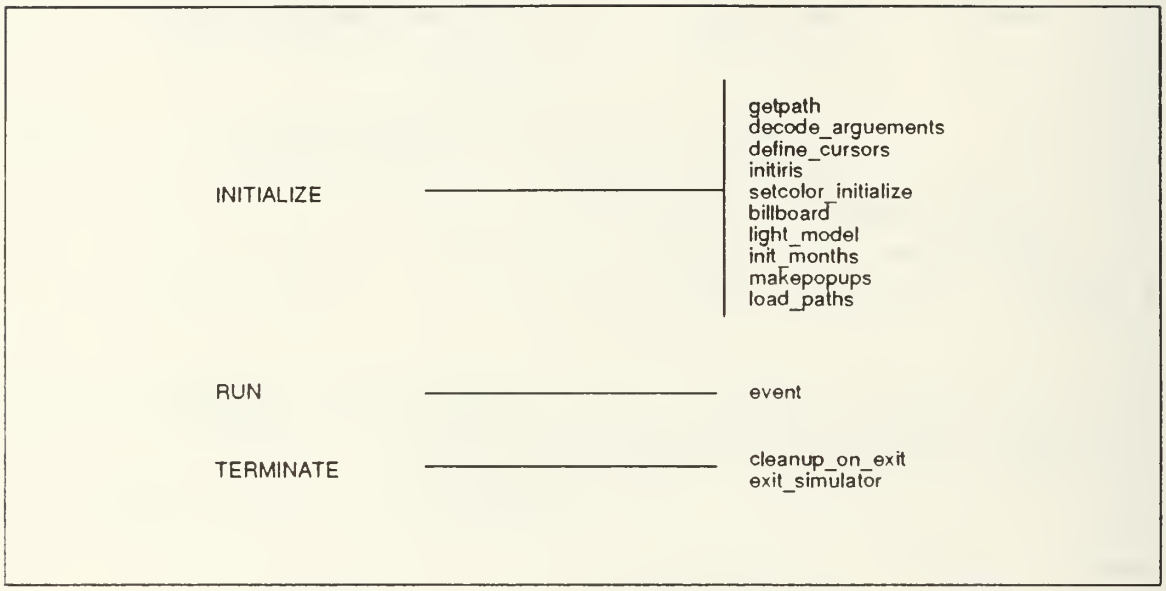

Figure A-3 MAIN Module Program Flow

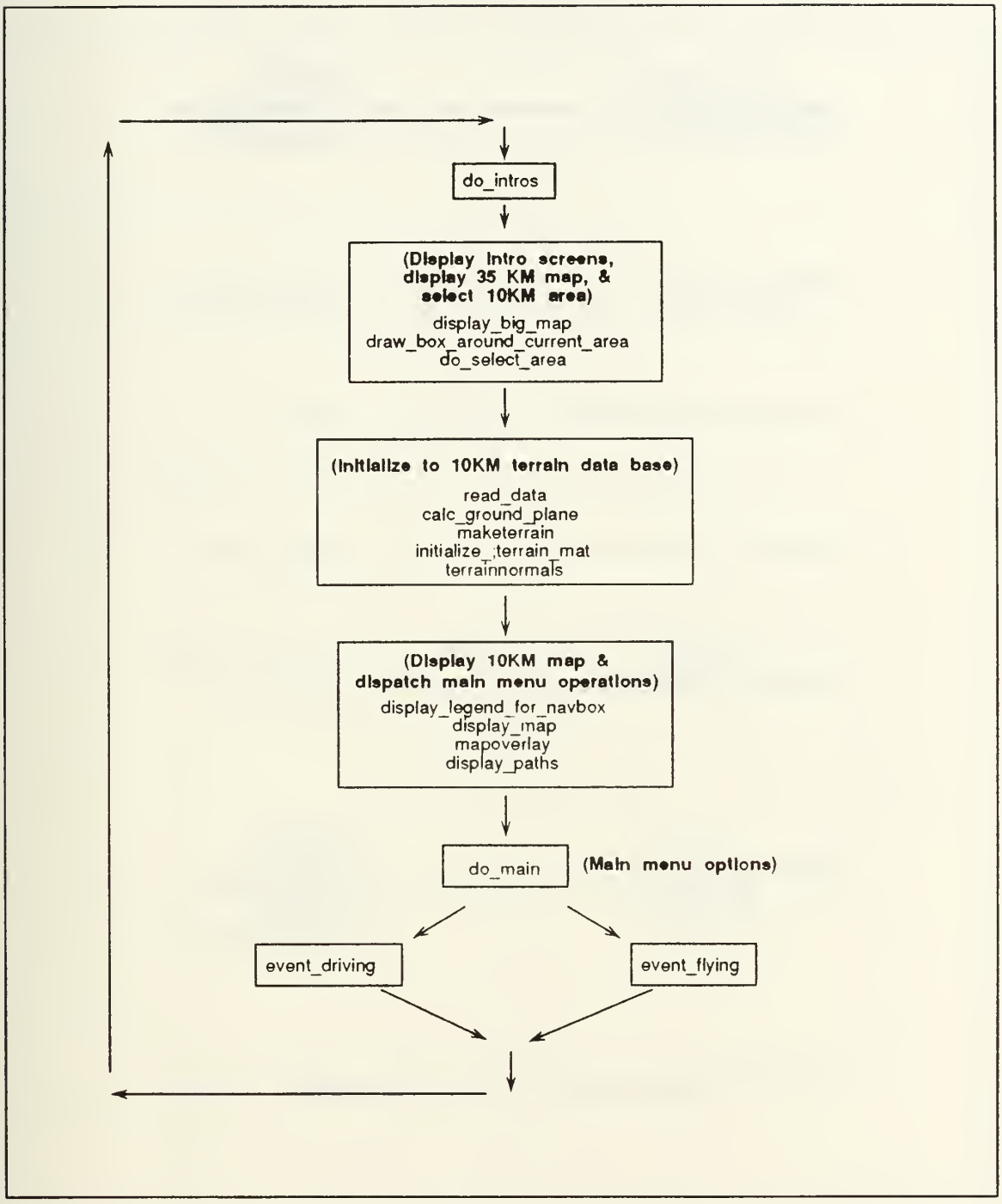

Figure A-4 Module event() Control Flow

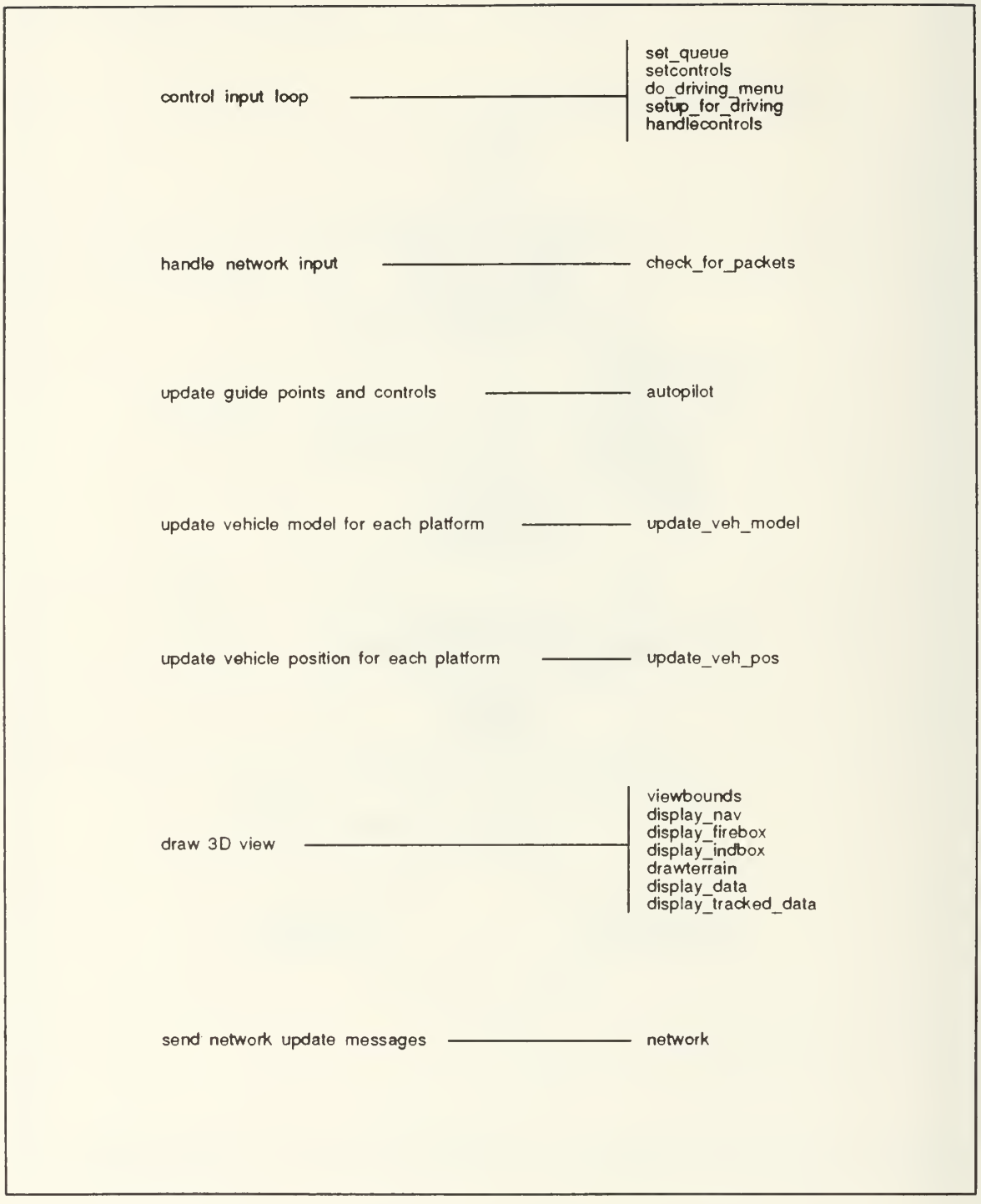

Figure A-5 Display Loop in event_driving

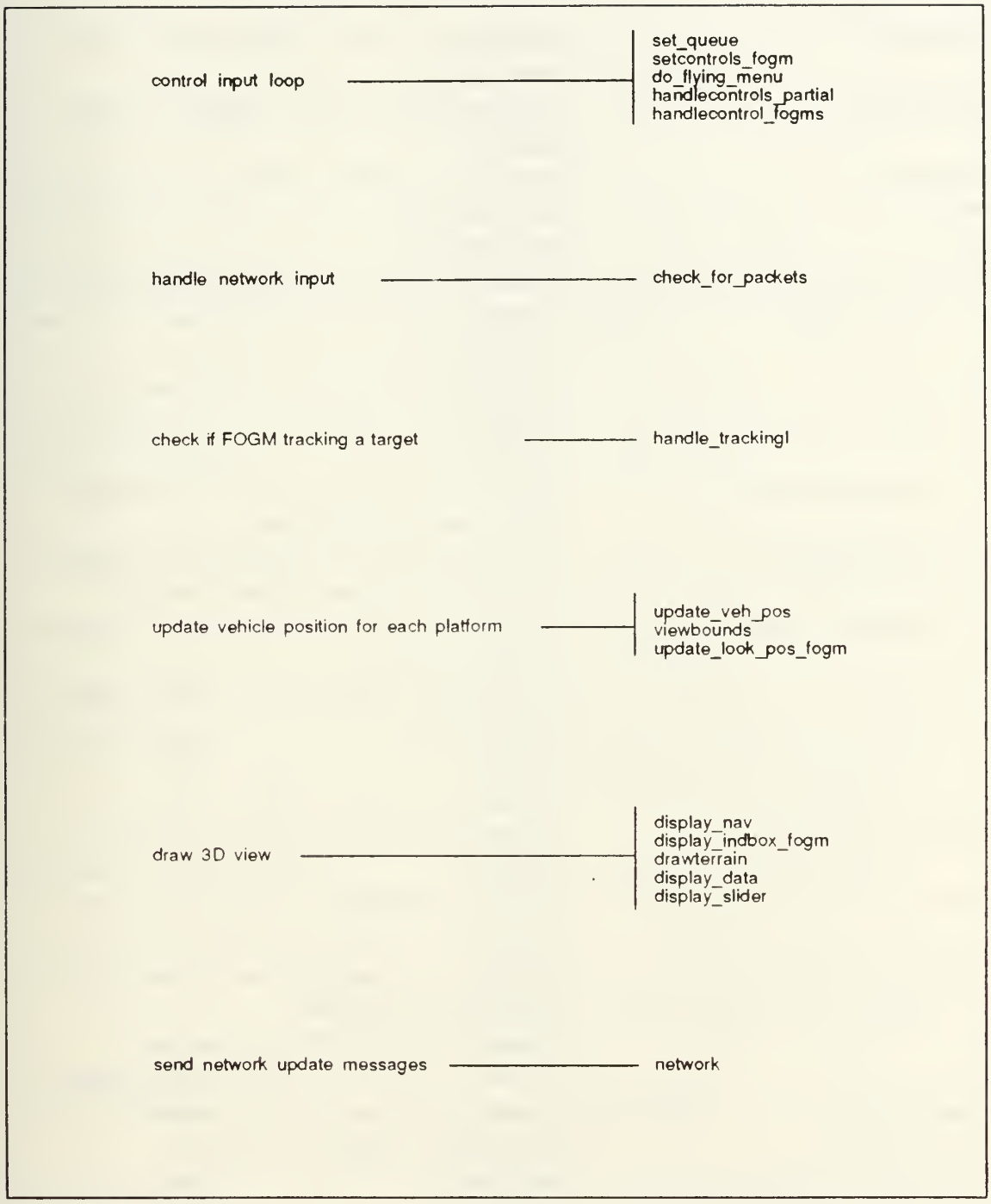

Figure A-6 Display Loop in event_flying

TABLE A-2 SUPPORT FUNCTIONS

addflash.c	Adds round impact flash event to the event list
addmessage.c	Adds message display event to event list
addsplash.c	Adds round splash event to event list
addveh.c	Adds platform
aps.c	Main routine
arcsine.c	Returns arcsine of input parameters
autopilot.c	Computes course and speed for platform(s)
billboard.c	Displays rotating billboard screen
bounce.c	Computes oscillation due to terrain irregularities
broadcast_services.c	Contains low-level network routines for broadcast messages
calc_eye_offset.c	Calculates world position of viewer
calc_ground_plane.c	Place ground plane under terrain
calc_look_parameters.c	Calculates viewer position and view point-of-aim
calcwindows.c	Calculates windows sizes and stores in an array
center_cursor.c	Positions cursor in the center of a window
center_string_map.c	Prints a centered string in the MAP window
center_string_menu.c	Prints a centered string in the MENU window
check_for_packets.c	Handles the reception and processing of network messages
check_round_in_flight.c	Updates round position, handles round impact with platform or ground
clearwindow.c	Clears a window to input color
cobra_normals.c	Calculates normals for Cobra helicopter
collision_detection.c	Detects collision between any two platforms
compute_slope.c	Computes the slope of a line
compute_start_stop.c	Computes information for drawterrain

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

compute_sun_location.c	Computes sun (light source) location based on month and hour
compute_x_bounds.c	Computes x drawing limit for drawterrain
compute_z_bounds.c	Computes z drawing limit for drawterrain
convert_to_dec_hr.c	Converts to decimal hour
convert_to_hr_min.c	Converts to hours and minutes
decode_arguments.c	Handle command line arguments when aps in started
define_cursors.c	Sets up cursor shapes
delete_veh.c	Deletes a platform and frees space
display_big_map.c	Displays 35KM 2D map
display_data.c	Displays current system parameters to user
display_elapsed_time.c	Converts floating point seconds into HMS formatted string
display_firebox.c	Displays mouse legend for platform with weapon system active
display_icon.c	Displays all platform icons
display_indbox.c	Displays mouse legend for platform without weapon system active
display_indbox_fogm.c	Displays mouse legend for FOGM platform
display_intro_screen.c	Displays program instructions
display_legend_for_big_map.c	Displays color gradation for elevation on 35KM map
display_legend_for_navbox.c	Displays color gradations for slope-colored 2D 10KM map used for path planning
display_map.c	Draws 10KM 2D map
display_nav.c	Draws blue course arrow and field-of-view limits
display_slider.c	Displays tracking controls for FOGM platform
display_tracked_message.c	Displays tracking message on the screen

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

do_capture.c	Handles storing platforms into data file
do_change_speed.c	Allows user to set the speed of all platforms
do_char.c	Displays a character in file name window
do_driving_menu.c	Displays driving menu and handles selection
do_flying_menu.c	Displays flying menu and handles selection
do_intros.c	Handles selection to display user instruction window
do_main.c	Builds and displays main menu and handles selection
do_main_reset.c	Clears all windows and displays 2D terrain map
do_pathops.c	Builds and displays path operations menu and handles selections
do_quitting.c	Handles program exit selection from any menu
do_resize.c	Handles resize selection from any menu
do_select_area.c	Handles menu selection of a 10KM operational area on the 35KM map
do_the_add.c	Handles menu selection of adding a platform
do_the_defaults.c	Handles menu selection of adding a default set of platforms
do_the_delete.c	Handles menu selection of deleting one or all platforms
do_the_select.c	Handles the selection of a platform
draw_box_around_current_area.c	Draws red box around current 10KM area on large map
draw_cobra.c	Draws the main body of the attack helicopter
draw_guidept.c	Draws a guide point as a marker on the terrain

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

draw_in_cobra.c	Draws the cockpit framework when looking from inside the attack helicopter
draw_main_rotor.c	Draws attack helicopter main rotor blade
draw_projectile.c	Draws round in flight
draw_reticle.c	Draws weapon sight picture
draw_tail_pipe.c	Draws attack helicopter IR suppressor
draw_tail_rotor.c	Draws attack helicopter tail rotor
drawflame.c	Draws flame from tail of FOGM
drawflash.c	Draws flash when round impacts a platform
drawgridbox.c	Draws a box in the map window
drawgun.c	Draws the tank barrel and bore evacuator
drawicon.c	Draws the icon for each type of platform
drawjeep.c	Draws the jeep
drawmissile.c	Draws the FOGM
drawopenjeep.c	Draws the open jeep
drawroller.c	Draws the tank rollers
drawsplash.c	Draws the ground splash when a projectile impacts the ground
drawtank.c	Draws the body of the tank
drawterrain.c	Main terrain and platform drawing routine
drawtire.c	Draws a tire
drawtrack.c	Draws a tank track
drawtruck.c	Draws the truck body
drawturit.c	Draws the tank turret
drawwreck.c	Draws a burning wreck
error_handler.c	Centralized error handler, just prints error message and returns
event.c	Main drawing cycle dispatch routine
event_driving.c	Ground platform drawing cycle
event_flying.c	FOGM drawing cycle
exit_simulator.c	Cleans up on exit

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

explosion.c	Flashes screen when current platform is destroyed
fire_blast.c	Flashes screen when weapon fires
fire_weapon.c	Handles weapon firing
flamenormals.c	Computes normals for the FOGM flame
gen_wildman_defaults.c	Generates a default set of platforms
get_curr_fps.c	Calculates current drawing rate in frames per second
get_mouse_xy.c	Gets current location of mouse cursor
get_name.c	Opens window for user to enter file name
gnd_level.c	Computes ground level of input *world* coordinates
gnd_level_UTM.c	Computes ground level of input *UTM* coordinates
guidance.c	Contains routines to handle transition between guidance states
gunnormals.c	Computes normals for tank barrel
handle_crash.c	Handles collision of two platforms
handle_events.c	Event handler package
handle_tracking.c	Handles FOGM tracking ground platform
handlecontrols.c	Handles mouse and dial inputs when driving a ground platform
handlecontrols_fogm.c	Handles dial inputs when flying the FOGM
handlecontrols_partial.c	Handles mouse inputs when flying the FOGM
highlitegrid.c	Highlights the 1 X 1 KM grids that contain any platforms for zooming
init_fonts.c	Initializes fonts and scales them to window size
init_months.c	Initialize month and lighting
init_network.c	Set up network sockets and stream server connection queue
init_weapons.c	Initializes any weapon systems on board a platform
initialize_terrain_mat.c	Defines materials for terrain polygons based on current color map

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

initiris.c	Initializes graphics system
initveh.c	Adds platforms from a file
jeepnormals.c	Computes normals for a jeep
letter.c	Draws a letter on the billboard
light_model_initialize.c	Initializes lighting model and lighting viewer definition
lightdefs.c	Defines materials, lights, and lighting model
limit_cursor_pick.c	Limits cursor for targeting attempt by FOGM
limit_value.c	Limits value between upper and lower bound
loadunit.c	Loads a unit matrix onto the stack
makepopups.c	Builds static menus
maketank.c	Builds polygon arrays for tank
maketerrain.c	Fills the terrain elevation and terrain polygon normal arrays
maketrack.c	Makes tank track polygons
mapoverlay.c	Draws the platform icons on the 2D 10KM map
math_utility.c	Package of math utility functions
missilenormals.c	Calculates normals for the FOGM missile
mousescreentoutm.c	Converts screen (pixel) coordinates to UTM coordinates
mousescreentoworld.c	Converts from screen coordinates to world graphics coordinates
mouseterraintoutm.c	Converts from 10KM coordinates to UTM coordinates
mouseutmtoscreen.c	Converts from UTM coordinates to point on the screen
mouseutmtoterrain.c	Converts from UTM coordinates to 10KM coordinates
mouseworldtoscreen.c	Converts from 2D world coordinates to screen coordinates
netstream_services.c	Package containing routines to manage stream connections
network.c	Builds messages and sends them

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

network_IO.c	Package of message level network communication routines
normalorient.c	Computes normal and reorganizes vertices of polygons
npoly_orient.c	Orients polygon vertices for backface method of hidden surface removal
obstacles.c	Stub module for obstacles package
openjeepnormals.c	Computes normals for open jeep
path.c	Package of routines to manage paths
placewindow_sizes.c	Sets aspect and size for billboard window
placewindows.c	Calculates the position of all windows and opens them
popwindow.c	Pops a window into full view
positionwindows.c	Positions windows under window manager
range_finder.c	Simulates a laser rangefinder and calculates the range to nearest platform in weapon sight crosshairs
read_data.c	Reads elevation and vegetation data from file
reset_tiltf.c	Resets FOGM tilt angle after releasing tracking mode
reticles.c	Variable definitions for sight reticle arrays
ring_the_bell.c	Rings the terminal bell
rollernormals.c	Computes normals for tank rollers
select_an_area.c	Handles selection of an area on 35KM map
select_grid_square.c	Handles selection of 1 X 1 KM grid square
select_sight.c	Displays viewing mode menu and handles selection
set_driven_view.c	Sets viewing parameters for perspective and eye geometry
set_popup_color.c	Sets the color of the popup menus
set_queue.c	Queues up dials and mouse

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

set_unqueue.c	Unqueues dials and mouse
setcolor.c	Sets current RGB color based on values in RGB color array
setcolor_initialize.c	Initializes RGB color array
setcontrols.c	Sets up controls for driving
setcontrols_fogm.c	Sets controls for flying the FOGM missile
setcursorcolor.c	Sets the current color of the cursor
setup_for_driving.c	Sets up for driving using mouse joystick
setup_navwin.c	Draws small 2D 10KM map in navigation window
setwindow.c	Puts focus into a window
setworldcoord.c	Saves world coordinates of window
simtime.c	Package containing routines to manage simulation time
sincos.c	Returns interpolated table lookup values for sine and cosine functions
switch_veh.c	Returns pointer to platform selected with mouse
tanknormals.c	Computes normals for a tank
terrainnormals.c	Computes normals for the terrain and stores them in an array
tirenormals.c	Computes normals for a tire
tot_num_ground_veh.c	Returns total number of ground platforms
tot_num_veh.c	Returns total number of platforms
tracking_check.c	Performs check of FOGM tracking system
tracknormals.c	Computes normals for tank tracks
trucknormals.c	Computes normals for truck
turitnormals.c	Computes normals for tank turret
update_look_pos.c	Calculates position that viewer is looking at for ground platform
update_look_pos_fogm.c	Calculates position that viewer is looking at for FOGM missile
update_veh_pos.c	Moves platform to new position
vecdotp.c	Computes vector dot product

TABLE A-2 SUPPORT FUNCTIONS - CONTINUED

vecmag.c	Computes magnitude of a vector
vehmodel.c	Package containing vehicle motion modeling routines
viewbounds.c	Computes viewing limits for drawterrain

APPENDIX B PATH PLANNER CODE

```
;;; -*- Mode: LISP; Package: USER; Syntax: Common-lisp -*-
;Title: clock functions
;Author: Shannon
;Date: 12 Apr 1989
;Discription: This program provides for the timming of clocks used in the Path Planner Control Program

(defflavor myclock ((start-iris-time 0)
                    (start-sym-time 0)
                    (last-iris-time 0)
                    (last-sym-time 0)
                    (delta-time 0)
                    )
        ()
        :initable-instance-variables)

(defmethod (:set-start-time myclock) (iris-time)
 (let* ()
  (setf last-iris-time iris-time)
  (setf start-iris-time iris-time)
  (setf start-sym-time (zl:time))
  (setf last-sym-time start-sym-time)
  (setf delta-time 0)
  )
 )

(defmethod (:reset-last-time myclock) (iris-time)
 (let* ((delta1 0.0) (delta2))
  (progn
   (setf last-sym-time (zl:time))
   (setf last-iris-time iris-time)
   (setf delta1 (- iris-time start-iris-time))
   (setf delta2 (- last-sym-time start-sym-time))
   )
  )
 )

(defmethod (:get-time myclock) ()
 (+ delta-time (/ (+ 0.0 (time-difference (zl:time) last-sym-time)) 60) last-iris-time)
 )

(defmethod (:get-all-times myclock) ()
 (princ start-iris-time)
 (princ start-sym-time)
 (princ last-iris-time)
```

```
  (princ last-sym-time)
  (princ delta-time)
)
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-
;Title: chaosflavor.lisp
;Author: Kwak
;Modified by: Shannon
;Date: 19 Apr 1989
;Discription:This code performs the communications between Symbolics computers using a character stream.


(load "comm-functions")

(defflavor mychaos ((host-name 'sym1)
                    (contact-name "user-chaos")
                    (contact nil)
                    (userstream nil)
                   )
        ()
       :initable-instance-variables)


(defmethod (:set-host-name mychaos)
           (name-of-host)
 (setf host-name name-of-host))

(defmethod (:set-contact-name mychaos) (name)
 (setf contact-name name))

(defmethod (:set-contact mychaos) (con)
 (setf contact con))

(defmethod (:set-stream mychaos) (str)
 (setf userstream str))

(defmethod (:start-user mychaos) (hostname contactname)
 (progn
  (send self :set-host-name hostname)
  (send self :set-contact-name contactname)
  (send self :set-contact (chaos:connect hostname contactname 13 72000))
  (send self :set-stream (chaos:make-stream contact :direction :bidirectional))
  (terpri)
  (princ "host name "   ) (princ host-name)
  (terpri)
  (princ "contact name ") (princ contact-name)
  (terpri)
 "A conversation using chaos has been established"))

(defmethod (:start-server mychaos) (contactname)
 (progn
```

```
    (send self :set-contact-name contactname)
    (send self :set-contact (chaos:listen contactname))
    (chaos:accept contact)
    (send self :set-stream (chaos:make-stream contact :direction :bidirectional))
    (terpri)
    (princ "host name "   ) (princ host-name)
    (terpri)
    (princ "contact name ") (princ contact-name)
    (terpri)
   "A conversation using chaos has been established"))

(defmethod (:put mychaos)
                (object)

  (send userstream :line-out object)
  (send userstream :force-output)
  )

(defun read-string-sym (stream num-chars)
  (let ((out-string ""))
    (dotimes (i num-chars)
      (setf out-string (string-append out-string (read-char-no-hang stream)))
      )
    out-string
    )
  )

(defmethod (:check-sym mychaos) (size-io)
  (let* ((typebuffer )
           )
    (progn
      (setq typebuffer
       (read-string-sym userstream size-io))
      )
    )
  )

(defmethod (:put-ready mychaos)
   (object)
;From path-planner to art
  (let* ((buffer "!!!!"))
    (setf buffer (string-append (string-append buffer object) "!!!"))
    (progn
      (send userstream :line-out buffer)
      (send userstream :force-output)
      't
      )
```

```
  )
 )

(defmethod (:put-waypoint mychaos)
                (object)
;from path-planner to art
 (let* ((buffer "@@@@"))
  (setf buffer (string-append
                (string-append buffer (convert-number-to-string object))
                              "@@@"))
  (progn
   (send userstream :line-out buffer)
   (send userstream :force-output)
   't
    )))

(defmethod (:load-map mychaos)
             (utm-e utm-n veh-id)
;From art to path planner
 (let* ((buffer "!!!!"))
  (setf buffer (string-append
   (string-append buffer
                 (convert-number-to-string (+ (* utm-e 1000000000000000)
                                              (* utm-n 10000000000)
                                              veh-id
                                               )
                                           )
                                       )
                             "!!!"
                              )
          )
  (progn
   (send userstream :line-out buffer)
   (send userstream  force-output)
   't
)))

(defmethod (:put-path mychaos)
  (org-utm-e org-utm-n start-utm-e start-utm-n goal-utm-e goal-utm-n veh-id)
;from art to path-planner
 (let* ((buffer "@@@@")
        (string-org-e (convert-number-to-string org-utm-e))
        (string-org-n (convert-number-to-string org-utm-n))
        (string-start-e (convert-number-to-string start-utm-e))
        (string-start-n (convert-number-to-string start-utm-n))
        (string-goal-e (convert-number-to-string goal-utm-e))
        (string-goal-and-id (convert-number-to-string
```

```
                    (+ (* goal-utm-n 10000000000)
                       veh-id
                       )
                    )
                   )
      )
    (setf buffer (string-append
                  (string-append
                   (string-append
                    (string-append
                     (string-append
                      (string-append
                       (string-append buffer
                                      string-org-e
                                      )
                                     string-org-n
                                     )
                                    string-start-e
                                    )
                                   string-start-n
                                   )
                                  string-goal-e
                                  )
                                 string-goal-and-id
                                 )
                                "@@@"
                                )
                  )
    (progn
     (send userstream :line-out buffer)
     (send userstream :force-output)
     't
     )
    )
  )


(defmethod (:stop mychaos)
           ()
    (send userstream :close :abort))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-
;Title: irisflavor3.lisp
;Author: Kwak
;Modification Author: Shannon
;Modification Date: 20 May 1989
;Discription: This code provides communications  functions to the symbolics workstation, whereby it can
communicate to the Iris

(defmacro loopfor (var init test exp1 &optional exp2 exp3 exp4 exp5)
 '(prog ()
      (setq ,var ,init)
      tag
        ,exp1
        ,exp2
        ,exp3
        ,exp4
        ,exp5
        (setq ,var (1+ ,var))
        (if (= ,var ,test) (return t) (go tag))))

(load "comm-functions")

(defvar *iris-port1* 1061)                    ; this is the send port
(defvar *iris-port2* 1061)                    ; this is the receive port
(defvar *local-talk-port* 1500)                 ; this is the local send
port
(defvar *local-listen-port* 1501)               ; this is the local
receive port



(defflavor conversation-with-iris ((talking-port-number     *iris-port1*)
                        (listening-port-number   *iris-port2*)
                        (local-talk-port-number  *local-talk-port*)
                        (local-listen-port-number
*local-listen-port*)
                        (talking-stream)
                        (listening-stream)
                        (destination-host-object)
                       )
                       ()
                       :initable-instance-variables)


(defmethod (:init-destination-host conversation-with-iris)
        (name-of-host)
  (setf destination-host-object (net:parse-host name-of-host)))
```

```
(defmethod (:start-ins conversation-with-iris) ()
 (setf talking-stream
      (tcp:open-tcp-stream destination-host-object
                  talking-port-number
                  local-talk-port-number))
 (setf listening-stream
      (tcp:open-tcp-stream destination-host-object
                  listening-port-number
                  local-listen-port-number))
 "A conversation with the ins machine has been established")

(defun read-string (stream num-chars)
 (let ((out-string ""))
  (dotimes (i num-chars)
   (setf out-string (string-append out-string (read-char-no-hang stream))))
  out-string))

(defmethod (:check-ins conversation-with-iris) (size-io)
 (let* ((typebuffer)
      )
  (progn
   (setf typebuffer
      (read-string listening-stream size-io)
      )
   )
  )
 )

(defvar *step-var* 0)


(defun my-write-string(string stream)
 (let* ((num-chars (length string)))
  (dotimes (i num-chars)
   (write-char (aref string i) stream)
   )
  )
 )

(defmethod (:put-waypoint conversation-with-ins)
      (veh-id utm-e utm-n)

 (let* ((buffer (string-append
           "//// "
```

```
            (string-append
              (convert-number-to-string veh-id)
              (string-append
                " "
                (string-append
                  (convert-number-to-string utm-e)
                  (string-append
                    " "
                    (string-append
                      (convert-number-to-string utm-n)
                      "////"
                      )
                    )
                  )
                )
              )
            )
          )
      (buffer-length  (length buffer))

      (lengthbuffer   (convert-number-to-string buffer-length))

      )

    (progn
      (my-write-string buffer talking-stream)
      (send talking-stream :force-output)
      )
    )
  )

(defmethod (:stop-iris conversation-with-iris)
        ()
  (progn (send talking-stream :close)
      (send listening-stream :close)))

?
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-
;title comm functions
;author Kwak
;discription This program provides functions to the communications progarams that convert to and from strings
;                    and numbers.


(defun convert-number-to-string (n)
  (princ-to-string n))

(defun convert-string-to-integer (str &optional (radix 10))
  (do ((j 0 (+ j 1))
       (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
      ((= j (length str)) n)))

(defun find-period-index (str)
  (catch 'exit
    (dotimes (x (length str) nil)
      (if (equal (char str x) (char "." 0))
      (throw 'exit x)))))

(defun get-leftside-of-real (str &optional (radix 10))
  (do  ((j 0 (1+ j))
       (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
       ((or (null (digit-char-p (char str j) radix)) (= j (length str)))  n)))

(defun get-rightside-of-real (str &optional (radix 10))
  (do ((index (1+ (find-period-index str)) (1+ index))
       (factor 0.10 (* factor 0 10))
       (n 0.0 (+ n (* factor (digit-char-p (char str index) radix)))))
      ((= index (length str)) n )))

(defun convert-string-to-real (str &optional (radix 10))
  (+ (float (get-leftside-of-real str radix)) (get-rightside-of-real str radix)))
```

```
;;; -*- Mode: LISP; Package: USER; Syntax: Common-lisp -*-
;Title define-packege-interface
;Author Shannon
;Date 24 May 1989
;Discription Defines the interface between programs running in different Symbolics packages

#L(defpackage cl-user
   (:export
     conversation-with-iris
     mychaos
     myclock
     convert-number-to-string
     convert-string-to-real
     convert-string-to-integer
     string-append
     princ-to-string))
```

```
;;; %%%  -*- Mode: ART; Syntax: Common-lisp; Base: 10.; Package: ART-USER -*-
;Title: Path Planner Control Program
;author: Shannon
;Date: 11 June 1989
;Discription: This program provides the over all control logic for finding a path and the sending that path to
;                the vehicle simulation.

#L(load "irisflavor3")
#L(load "chaosflavor")
#L(load "def-interface")
#L(load "clockflavor")
#L(defvar talk-s)
#L(defvar talk-i)
#L(setf talk-i (scl:make-instance 'user:conversation-with-ins))
#L(setf talk-s (scl:make-instance 'user:mychaos))

(defschema counter
        (seq 0)
        )

(defschema obsticle
        (instance-of counter)
        (nw-utm-e  000)
        (nw-utm-n  000)
        (sw-utm-e  000)
        (sw-utm-n  000)
        (se-utm-e  000)
        (se-utm-n  000)
        (ne-utm-e  000)
        (ne-utm-n  000)
        (seq      000)
        (seq-ord   last)
        )

(defschema obj-type
        (type  unk)
        )

(defschema location
        (utm-e 000)
        (utm-n 000)
        )

(defschema id
        (veh-id     000)
        )
```

```
(defschema map-state
        (ready      not-yet)
        )

(defschema clock
        (clock-id   new)
        (time       000)
        )

(defschema control-conditions
        (new-goal    no)
        (quit-all    no)
        (broke-down  no)
        (pause       no)
        (whole-path  no)
        (new-time    no)
        (new-waypoint no)
        (old-time    0)
        )

(defschema initial-map-points
        (org-utm-e   000)
        (org-utm-n   000)
        (start-utm-e 000)
        (start-utm-n 000)
        (goal-utm-e  000)
        (goal-utm-n  000)
        )

(defschema control
        (instance-of  clock)
        (instance-of  obj-type)
        (instance-of  id)
        (instance-of  counter)
        (instance-of  control-conditions)
        )

(defschema map
        (instance-of obj-type)
        (instance-of location)
        (instance-of id)
        (instance-of map-state)
        )

(defschema init
        (instance-of obj-type)
        (instance-of id)
```

```
        (instance-of initial-map-points)
        )

(defschema veh
        (cse        0)
        (vel        0)
        (guide      0)
        )

(defschema veh-change
        (delta-time  0)
        (new-position no)
        )

(defschema msg-state
        (current no)
        )

(defschema veh-state
        (instance-of  obj-type)
        (instance-of  veh)
        (instance-of  id)
        (instance-of  location)
        (instance-of  veh-change)
        )

(defschema veh-msg
        (instance-of clock)
        (instance-of obj-type)
        (instance-of veh)
        (instance-of id)
        (instance-of location)
        (instance-of msg-state)
        )

(defschema machine-type
        (one   one)
        (two   two)
        (three three)
        (four  four)
        (five  five)
        )

(defschema sym
        (instance-of machine-type)
        (one        sym1)
        (two        sym2)
```

```
        (three      sym3)
        )

(defschema ins
        (instance-of machine-type)
        (one        gravy1)
        (two        gravy2)
        (three      gravy3)
        (four       gravy4)
        (five       gravy5)
        )

(defrelation msg-sym (?type))

(defrelation msg-ins (?type))

(defrelation start-ins-comm (?t-or-f))

(defrelation start-sym-comm (?t-or-f))

(defrelation menu (?one-or-two))

(defrelation sym-on (?yes))

(defrelation check-comm (?ins-and-sym))

(defrelation clock-update (?yes))

(defrelation sym-link (?code))

(defrelation iris-link (?code))

(deffacts initialization
        (menu one)
        )

(defrule menu1
       (declare (salience -1000))
  (schema sym
        (one   ?s1)
        (two   ?s2)
        (three ?s3)
        )
  ?a <- (menu one)
  =>
  (printout t t "Where is the path planner located?")
  (printout t t "Your choices are the following, chose one by it's letter. "
```

```
        t "a " ?s1
        t "b " ?s2
        t "c " ?s3
        t "NOTE---Please ensure that the path planning software is running"
        t)
  (bind ?b (read))
  (if (or (eq ?b 'a)
          (eq ?B 'A))
      then
      (assert (sym-link ?s1)
            (menu two)
            (start-sym-comm yes)
            )
      else
      (if (or (eq ?b 'b)
            (eq ?b 'B))
          then
          (assert (sym-link ?s2)
                (menu two)
                (start-sym-comm yes)
                )
          else
          (if (or (eq ?b 'c)
                (eq ?b 'C))
              then
              (assert (sym-link ?s3)
                    (menu two)
                    (start-sym-comm yes)
                    )
              else
              (retract ?a)
              (assert (menu one))
              )
          )
      )
  (retract ?a)
  )

(defrule menu2
        (declare (salience -1000))
  (schema iris
        (one   ?i1)
        (two   ?i2)
        (three ?i3)
        (four  ?i4)
        (five  ?i5)
        )
```

```
?a <- (menu two)
=>
(printout t t "Where is the vehicle simulator located?")
(printout t t "Your choices are the following, chose one by it's letter. "
        t "a " ?i1
        t "b " ?i2
        t "c " ?i3
        t "d " ?i4
        t "e " ?i5
        t "NOTE---Please ensure that the simulator is running"
        t)
(bind ?b (read))
(if (or (eq ?b 'a)
      (eq ?b 'A))
   then
   (assert (iris-link ?i1))
   (assert (start-iris-comm yes))
   else
   (if (or (eq ?b 'b)
         (eq ?b 'B))
      then
      (assert (iris-link ?i2))
      (assert (start-iris-comm yes))
      else
      (if (or (eq ?b 'c)
            (eq ?b 'C))
         then
         (assert (iris-link ?i3))
         (assert (start-iris-comm yes))
         else
         (if (or (eq ?b 'd)
               (eq ?b 'D))
            then
            (assert (iris-link ?i4))
            (assert (start-iris-comm yes))
            else
            (if (or (eq ?b 'e)
                  (eq ?b 'E))
               then
               (assert (iris-link ?i5))
               (assert (start-iris-comm yes))
               else
               (retract ?a)
               (assert (menu two))
               )
            )
         )
```

```
        )
      )
  (retract ?a)
  )

(defrule start-iris-comm-links
        (declare (salience -1000))
  (iris-link ?iris-machine)
  ?a <- (start-iris-comm yes)
  =>
  #L(scl:send talk-i :init-destination-host ?iris-machine)
  #L(scl:send talk-i :start-iris)
  (retract ?a)
  (assert (check-comm iris)
        )
  )

(defrule start-sym-comm-links
        (declare (salience -1000))
  (sym-link ?sym-machine)
  ?a <- (start-sym-comm yes)
  =>
  #L(scl:send talk-s :start-user ?sym-machine "path")
  (retract ?a)
  (assert (sym-on    yes)
        )
  )

(defrule check-comm-links-iris
        (declare (salience 500))
  ?a <- (check-comm iris)
  =>
  (bind ?b #L(scl:intern (scl:send talk-i :check-iris 1)))
  (if (eq ?b NIL) then
     (retract ?a)
     (assert
       (check-comm iris)
       (check-comm sym)
       (clock-update yes)
       )
     else
     (retract ?a)
     (assert (msg-iris ?b))
     )
  )
```

```
(defrule check-comm-links-sym
      (declare (salience 500))
 (schema ?any
       (or (ready sent)
          (ready ready)
          )
       )
 ?a <- (check-comm sym)
 =>
 (bind ?b #L(scl:intern (scl:send talk-s :check-sym 1)))
 (if (eq ?b NIL) then
    (retract ?a)
    (assert (check-comm iris))
    else
    (retract ?a)
    (assert (msg-sym ?b))
    )
 )

(defrule read-update-in
      (declare (salience 1000))
 ?msg <- (msg-ins ?a)
 (test (eq ?a '>))
 =>
 (bind ?b #L(scl:intern (scl:send talk-i :check-iris 3)))
 (if (eq ?b '>>>) then
    (bind ?veh-id #L(scl:send talk-i :check-iris 10))
    (bind ?utm-e  #L(scl:send talk-i :check-iris 10))
    (bind ?utm-n  #L(scl:send talk-i :check-iris 10))
    (bind ?cse    #L(scl:send talk-i :check-iris 10))
    (bind ?vel    #L(scl:send talk-i :check-iris 10))
    (bind ?time   #L(scl:send talk-i :check-iris 10))
    (bind ?guide  #L(scl:send talk-i :check-iris 1))
    (bind ?b #L(scl:intern (scl:send talk-i :check-iris 3)))
    (if (eq ?b '>>>) then
       (bind ?msg-id "MSG")
       (bind ?msg-id #L(scl:intern (user:string-append ?msg-id ?veh-id)))
       (bind ?veh-id #L(user:convert-string-to-integer ?veh-id))
       (bind ?utm-e  #L(floor (user:convert-string-to-real ?utm-e)))
       (bind ?utm-n  #L(floor (user:convert-string-to-real ?utm-n)))
       (bind ?cse    #L(user:convert-string-to-real ?cse))
       (bind ?vel    #L(user:convert-string-to-real ?vel))
       (bind ?time   #L(user:convert-string-to-real ?time))
       (bind ?guide  #L(user:convert-string-to-integer ?guide))
       (modify (schema ?msg-id
                  (veh-id  ?veh-id)
                  (utm-e   ?utm-e)
```

```
                (utm-n   ?utm-n)
                (cse     ?cse)
                (vel     ?vel)
                (time    ?time)
                (guide   ?guide)
                (current yes)
                )
            )
        )
    )
 (retract ?msg)
 (assert (check-comm iris)
        (check-comm sym)
        (clock-update yes)
        )
 )

(defrule read-init-in
      (declare (salience 1000))
 ?msg <- (msg-iris ?a)
 (test (eq ?a '<))
 =>
 (bind ?b #L(scl:intern (scl:send talk-i :check-iris 3)))
 (if (eq ?b '<<<) then
    (bind ?org-utm-e    #L(scl:send talk-i :check-iris 10))
    (bind ?org-utm-n    #L(scl:send talk-i :check-iris 10))
    (bind ?veh-id       #L(scl:send talk-i :check-iris 10))
    (bind ?start-utm-e  #L(scl:send talk-i :check-iris 10))
    (bind ?start-utm-n  #L(scl:send talk-i :check-iris 10))
    (bind ?goal-utm-e   #L(scl:send talk-i :check-iris 10))
    (bind ?goal-utm-n   #L(scl:send talk-i :check-iris 10))
    (bind ?time         #L(scl:send talk-i :check-iris 10))
    (bind ?b #L(scl:intern (scl:send talk-i :check-ins 3)))
    (if (eq ?b '<<<) then
       (bind ?init "INIT")
       (bind ?init #L(scl:intern (user:string-append ?init ?veh-id)))
       (bind ?map "MAP")
       (bind ?map #L(scl:intern (user:string-append ?map ?veh-id)))
       (bind ?cont "CONTROL")
       (bind ?cont #L(scl:intern (user:string-append ?cont ?veh-id)))
       (bind ?veh "VEH")
       (bind ?veh #L(scl:intern (user:string-append ?veh ?veh-id)))
       (bind ?msg-id "MSG")
       (bind ?msg-id #L(scl:intern (user:string-append ?msg-id ?veh-id)))
       (bind ?org-utm-e   #L(floor (user:convert-string-to-real ?org-utm-e)))
       (printout t t ?org-utm-e)
       (bind ?org-utm-n   #L(floor (user:convert-string-to-real ?org-utm-n)))
```

```
(printout t t ?org-utm-n)
(bind ?veh-id     #L(user:convert-string-to-integer ?veh-id))
(bind ?start-utm-e #L(floor (user:convert-string-to-real
                    ?start-utm-e)))
(bind ?start-utm-n #L(floor (user:convert-string-to-real
                    ?start-utm-n)))
(bind ?goal-utm-e  #L(floor (user:convert-string-to-real
                    ?goal-utm-e)))
(bind ?goal-utm-n  #L(floor (user:convert-string-to-real
                    ?goal-utm-n)))
(bind ?time       #L(user:convert-string-to-real ?time))
(bind ?clock-id   #L(scl:make-instance 'user:myclock))
#L(scl:send ?clock-id :set-start-time ?time)
(assert (schema ?init
          (instance-of init)
          )
     (schema ?map
          (instance-of map)
          )
     (schema ?cont
          (instance-of control)
          )
     (schema ?veh
          (instance-of veh-state)
          )
     (schema ?msg-id
          (instance-of veh-msg)
          )
     )
(modify (schema ?init
          (org-utm-e   ?org-utm-e)
          (org-utm-n   ?org-utm-n)
          (veh-id      ?veh-id)
          (start-utm-e ?start-utm-e)
          (start-utm-n ?start-utm-n)
          (goal-utm-e  ?goal-utm-e)
          (goal-utm-n  ?goal-utm-n)
          (type        init)
          )
     (schema ?map
          (veh-id      ?veh-id)
          (utm-e       ?org-utm-e)
          (utm-n       ?org-utm-n)
          (ready       send)
          (type        map)
          )
     (schema ?cont
```

```
                (new-goal   yes)
                (time       ?time)
                (old-time   ?time)
                (veh-id     ?veh-id)
                (clock-id   ?clock-id)
                (type       cont)
                (seq        1)
                )
            (schema ?veh
                (veh-id     ?veh-id)
                (utm-e      ?start-utm-e)
                (utm-n      ?start-utm-n)
                (type       veh)
                )
            (schema ?msg-id
                (type   msg)
                (current no)
                )
            )
        )
    )
 (retract ?msg)
 (assert
       (check-comm sym)
       (clock-update yes)
       )
 )

(defrule process-map-loaded-msg
       (declare (salience 1000))
 ?msg <- (msg-sym ?a)
 (test (eq ?a '!))
 =>
 (bind ?b #L(scl:intern (scl:send talk-s :check-sym 3)))
 (if (eq ?b '!!!) then
    (bind ?cond   #L(scl:intern (scl:send talk-s :check-sym 5)))
    (bind ?veh-id #L(scl:send talk-s :check-sym 10))
    (bind ?b      #L(scl:intern (scl:send talk-s :check-sym 3)))
    (if (and (eq ?b '!!!)
             (eq ?cond 'READY)) then
           (bind ?map "MAP")
           (bind ?map #L(scl:intern (user:string-append ?map ?veh-id)))
           (modify (schema ?map
                    (ready ready)
                    )
                )
           )
```

```
   )
  (retract ?msg)
  (assert (check-comm iris))
  )

(defrule process-waypoint-in-msg
      (declare (salience 1000))
  ?msg <- (msg-sym ?a)
  (test (eq ?a '@))
  =>
  (bind ?b #L(scl:intern (scl:send talk-s :check-sym 3)))
  (if (eq ?b '@@@) then
    (bind ?utm-e   #L(scl:send talk-s :check-sym 5))
    (bind ?utm-n   #L(scl:send talk-s :check-sym 5))
    (bind ?veh-id  #L(scl:send talk-s :check-sym 10))
    (bind ?seq     #L(scl:send talk-s :check-sym 5))
    (bind ?b #L(scl:intern (scl:send talk-s :check-sym 3)))
    (if (eq ?b '@@@) then
      (bind ?way "WAYPOINT")
      (bind ?way #L(scl:intern (user:string-append
                        (user:string-append ?way
                                    ?veh-id
                                    )
                        ?seq
                        )
                       )
          )
      (bind ?utm-e   #L(floor (user:convert-string-to-integer ?utm-e)))
      (bind ?utm-n   #L(floor (user:convert-string-to-integer ?utm-n)))
      (bind ?veh-id  #L(floor (user:convert-string-to-integer ?veh-id)))
      (bind ?seq     #L(floor (user:convert-string-to-integer ?seq)))
      (assert (schema ?way
                (instance-of id)
                (instance-of counter)
                (instance-of obj-type)
                (instance-of location)
                (type    w-point)
                (utm-e   ?utm-e)
                (utm-n   ?utm-n)
                (veh-id  ?veh-id)
                (seq     ?seq)
                )
            )
      )
    )
  (if (0 = ?seq) then
    #L(scl:send talk-i :put-waypoint ?veh-id ?utm-e ?utm-n)
```

```
    )
  (retract ?msg)
  (assert (check-comm ins))
  )

(defrule clean-up-waypoints
      (declare (salience 6000))
  (schema ?way
       (type   w-point)
       (veh-id  ?veh-id)
       )
  (schema ?msg
       (type   msg)
       (veh-id  ?veh-id)
       (guide  0)
       (current yes)
       )
  (schema ?veh
       (veh-id     ?veh-id)
       (type       veh)
       (guide      1)
       )
  =>
  (retract
    (schema ?way
         (instance-of id)
         (instance-of counter)
         (instance-of obj-type)
         (instance-of location)
         )
   )
  )

(defrule clean-up-vehicle
      (declare (salience 5000))
  (schema ?msg
       (type   msg)
       (veh-id  ?veh-id)
       (guide  0)
       (current yes)
       )
  (schema ?init
       (veh-id     ?veh-id)
       (type       init)
       )
  (schema ?map
       (veh-id     ?veh-id)
```

```
      (type       map)
      )
 (schema ?cont
      (veh-id     ?veh-id)
      (type       cont)
      )
 (schema ?veh
      (veh-id     ?veh-id)
      (type       veh)
      (guide      1)
      )

 =>
 (retract
  (schema ?init
       (instance-of init)
       )
  (schema ?map
       (instance-of map)
       )
  (schema ?cont
       (instance-of control)
       )
  (schema ?veh
       (instance-of  veh-state)
       )
  )
 )

(defrule clean-up-sym-msg
      (declare (salience 500))
 ?msg <- (msg-sym ?code)
 =>
 (retract ?msg)
 (assert
  (check-comm sym)
  (check-comm iris)
  (clock-update yes)
  )
 )

(defrule clean-up-iris-msg
      (declare (salience 500))
 ?msg <- (msg-iris ?code)
 =>
 (retract ?msg)
 (assert
```

```
   (check-comm iris)
   (check-comm sym)
   (clock-update yes)
   )
  )

(defrule load-map
      (declare (salience 1000))
  (sym-on yes)
  (schema ?map
        (utm-e  ?org-e)
        (utm-n  ?org-n)
        (veh-id ?veh-id)
        (ready  send)
        (type   map)
        )
  =>
  #L(scl:send talk-s :load-map ?org-e ?org-n ?veh-id)
  (modify
   (schema ?map
         (ready sent)
         )
   )
  (assert (check-comm sym))
  )

(defrule start-path
      (declare (salience 5000))
  (schema ?veh
        (type   veh)
        (guide  1)
        (veh-id ?veh-id)
        )
  (schema ?init
        (org-utm-e    ?org-e)
        (org-utm-n    ?org-n)
        (start-utm-e  ?start-e)
        (start-utm-n  ?start-n)
        (goal-utm-e   ?goal-e)
        (goal-utm-n   ?goal-n)
        (type         init)
        (veh-id       ?veh-id)
        )
  (schema ?map
        (veh-id ?veh-id)
        (type   map)
        (ready  ready)
```

```
      )
 (schema ?control
      (new-goal yes)
      (veh-id   ?veh-id)
      (type     cont)
      )
 =>
 #L(scl:send talk-s :put-path ?org-e ?org-n ?start-e ?start-n ?goal-e ?goal-n
                    ?veh-id)
 (modify
  (schema ?control
       (new-goal no)
      )
  )
 )

(defrule send-new-waypoint
     (declare (salience 5000))
 (schema ?any
      (type   w-point)
      (veh-id ?veh-id)
      (seq    ?seq)
      (utm-e  ?east)
      (utm-n  ?north)
      )
 (schema ?control
      (seq      ?seq-num)
      (veh-id   ?veh-id)
      (type     cont)
      (new-waypoint yes)
      )
 (test (and (?seq-num < ?seq)
       (?seq-num + 3 > ?seq)
       )
    )
 =>
 #L(scl:send talk-i :put-waypoint ?veh-id ?east ?north)
 (modify (schema ?control
           (seq        ?seq)
           (new-waypoint no)
           )
      )
 (if (?seq-num = 1) then
   (assert (clock-update yes))
   )
 )
```

```
(defrule update-vehicle
      (declare (salience 1000))
 (schema ?veh-msg
       (type    msg)
       (veh-id  ?veh-id)
       (utm-e   ?utm-e)
       (utm-n   ?utm-n)
       (cse     ?cse)
       (vel     ?vel)
       (time    ?time)
       (guide   ?guide)
       (current yes)
       )
 (schema ?control
       (type   cont)
       (veh-id ?veh-id)
       )
 (schema ?veh-current
       (veh-id ?veh-id)
       (type   veh)
       )
 =>
 (modify (schema ?control
            (time         ?time)
            (new-time     yes)
            )
       (schema ?veh-current
            (utm-e       ?utm-e)
            (utm-n       ?utm-n)
            (cse         ?cse)
            (vel         ?vel)
            (guide       ?guide)
            (new-position yes)
            )
       (schema ?veh-msg
            (current no)
            )
       )
 )

(defrule update-clock
      (declare (salience 500))
 ?test <- (clock-update yes)
 (schema ?control
       (veh-id    ?veh-id)
       (time      ?time)
       (clock-id  ?clock-id)
```

```
        (type      cont)
        )
  (schema ?veh
        (veh-id      ?veh-id)
        (type        veh)
        (delta-time   ?delta-time)
        (new-position no)
        )
  (test (?delta-time = 0))
  =>
  (bind ?current-time #L(scl:send ?clock-id :get-time))
  (bind ?delta-time (?current-time - ?time))
  (modify (schema ?control
              (time ?current-time)
              )
        (schema ?veh
              (delta-time   ?delta-time)
              )
        )
  (retract ?test)
  )

(defrule reset-clock
       (declare (salience 5000))
  (schema ?control
        (time      ?time)
        (old-time   ?old-time)
        (clock-id   ?clock-id)
        (new-time   yes)
        (type      cont)
        )
  (schema ?veh
        (veh-id      ?veh-id)
        (type        veh)
        (delta-time   ?delta-time)
        (new-position ?no)
        )
  =>
  (if (eq ?no 'NO) then
     (bind ?delta-time (?time - ?old-time))
     )
  #L(scl:send ?clock-id :reset-last-time ?time)
  (modify (schema ?control
              (old-time   ?time)
              (new-time   no)
              )
        (schema ?veh
```

```
            (delta-time   ?delta-time)
            (new-position no)
            )
        )
 )

(defrule change-position
      (declare (salience 5000))
 (schema ?veh
        (type       veh)
        (utm-e      ?utm-e)
        (utm-n      ?utm-n)
        (cse        ?cse)
        (vel        ?vel)
        (delta-time   ?delta-time)
        (new-position no)
        )
 (test (?delta-time > 0))
 =>
 (bind ?delta-dist (?vel * ?delta-time))
 (bind ?utm-e #L(floor (+ ?utm-e (* ?delta-dist (cos ?cse)))))
 (bind ?utm-n #L(floor (+ ?utm-n (* ?delta-dist (sin ?cse)))))
 (modify (schema ?veh
            (utm-e  ?utm-e)
            (utm-n  ?utm-n)
            (delta-time   0)
            (new-position yes)
            )
        )
 )

(defrule new-waypoint
      (declare (salience 1000))
 (schema ?veh
        (type       veh)
        (veh-id     ?veh-id)
        (new-position yes)
        (utm-e      ?utm-e)
        (utm-n      ?utm-n)
        (guide      1)
        )
 (schema ?control
        (type   cont)
        (seq    ?seq)
        (veh-id ?veh-id)
        )
 (schema ?any
```

```
      (type   w-point)
      (veh-id ?veh-id)
      (seq    ?seq)
      (utm-e  ?east)
      (utm-n  ?north)
      )
=>
(if (200 > #L(let ((dx (- ?east ?utm-e))
                   (dy (- ?north ?utm-n))
                   )
              (sqrt (+ (* dx dx) (* dy dy)))
              )
      )
   then
   (modify
    (schema ?control
            (new-waypoint yes)
            )
    )
   )
(modify (schema ?veh
             (new-position no)
             )
      )
)
```

```
;;; -*- Mode: LISP; Package: USER; Syntax: Common-lisp -*-
;Title: Search Control Program
;Author: Shannon
;Date: 5 Jun 1989
;Discription: This program controls the flow of the search algorithm.


(defvar *done*)
(setf  *done* nil)
(defvar *maps*)
(setf  *maps* nil)
(defvar *vehs*)
(setf  *vehs* nil)
(defvar *wave-paths*)
(setf  *wave-paths* nil)

(defun convert-to-utm (node-list map-utm-e map-utm-n)
 (let ((num-nodes (length node-list)))
  (dotimes (x num-nodes)
   (cond
    ((eq x (- num-nodes 1))
     (setf node-list (cdr node-list))
     )
    (t
     (setf (car (car node-list)) (new-utm (car (car node-list)) map-utm-e))
     (setf (car (cdr (car node-list))) (new-utm (car (cdr (car node-list)))
                                        map-utm-n))
     (setf node-list (append (cdr node-list) (list (car node-list))))
     )
    )
   )
  node-list
  )
 )

(defun send-waypoints (wave-paths)
 (let ((num-waves (length wave-paths)))
  (dotimes (x num-waves)
   (terpri)
   (cond
    ((null (car wave-paths))
     (setf wave-paths (cdr wave-paths))
     )
    ((cdr (cdr (car wave-paths)))
     (send talk-s :put-waypoint
         (+ (* (car (cdr (cdr (car (car wave-paths)))))
             10000000000000000000
```

```
                )
                (* (car (cdr (cdr (cdr (car (car wave-paths))))))
                  10000000000000000
                  )
                (* (car (cdr (car (car wave-paths)))) 100000)
                (car (car (car wave-paths)))
                )
              )
            (setf wave-paths (append (cdr wave-paths)
                                     (list (cdr (car wave-paths)))
                                     )
              )
            )
           (t
            (send talk-s :put-waypoint
              (+ (* (car (cdr (cdr (car (cdr (car wave-paths))))))
                  100000000000000000000
                  )
                (* (car (cdr (cdr (cdr (car (cdr (car wave-paths)))))))
                  10000000000000000
                  )
                (* (car (cdr (car (cdr (car wave-paths))))) 100000)
                (car (car (cdr (car wave-paths))))
                )
              )
            (setf wave-paths (cdr wave-paths))
            )
           )
          )
         wave-paths
         )
        )

(defun add-id (node-list veh-id)
  (let ((num-nodes (length node-list)))
    (dotimes (x num-nodes)
      (setf node-list (append (cdr node-list)
                              (list (cons veh-id (car node-list)))
                              )
        )
      )
    node-list
    )
  )

(defun add-seq-num (node-list)
  (let ((num-nodes (length node-list)) (seq 0))
```

```
  (dotimes (x num-nodes)
   (setf node-list (append (cdr node-list)
                           (list (cons seq (car node-list)))
                           )
                   )
   (setf seq (+ seq 1))
   )
  node-list
  )
 )

(defun start-search-control
    ()
 (search-control)
 )

(defun search-control
    ()
 (load "lr-wave")
 (load "chaosflavor")
 (setf talk-s (make-instance 'mychaos))
 (send talk-s :start-server "path")
 (do* ((control-s (send talk-s :check-sym 1)
                  (send talk-s :check-sym 1)
                  )
       )
      ((setf time-to-quit "done")
       (send talk-s :stop)
       )
   (cond
    ((equal control-s "!")
     (setf next-3 (send talk-s :check-sym 3))
     (cond
      ((equal next-3 "!!!")
       (setf map-str-utm-e (send talk-s :check-sym 5))
       (setf map-str-utm-n (send talk-s :check-sym 5))
       (setf veh-id (send talk-s :check-sym 10))
       (setf next-3 (send talk-s :check-sym 3))
       (cond
        ((equal next-3 "!!!")
         (terpri)
         (princ "loading map")
         (setf map-utm-e (convert-string-to-integer map-str-utm-e))
         (setf map-utm-n (convert-string-to-integer map-str-utm-n))
         (setf veh-map (intern (string-append
                                (string-append "MAP"
                                               map-str-utm-e
```

```
                    )
                map-str-utm-n
                )
               )
    )
(setf current-veh (intern (string-append "VEH" veh-id)))
(do ((maps *maps* (cdr maps)))
  ((or (equal veh-map (car maps))
       (null (cdr maps))
       )
   (cond
    ((equal veh-map (car maps))
     (send talk-s :put-ready
         (string-append "READY"
                 veh-id
                 )
         )
     )
    ((null (cdr maps))
     (setf (symbol-value veh-map) (make-array '(102 102)))
     (setf *maps* (cons veh-map *maps*))
     (send talk-s :put-ready
         (string-append (load-map 100
                          map-utm-e
                          map-utm-n
                          "bin-slope.dat"
                          (symbol-value veh-map)
                          )
                 veh-id
                 )
         )
     )
    )
   )
 )
(setf (symbol-value current-veh) veh-map)
(do ((vehs *vehs* (cdr vehs)))
  ((or
    (eq current-veh (car vehs))
    (null (cdr vehs))
    )
   (cond
    ((null (cdr vehs)))
    (setf *vehs* (cons current-veh *vehs*))
    )
   )
 )
```

```
      (terpri)
      (princ "map loaded")
      )
     )
    )
   )
  )
 ((equal control-s "@")
  (setf next-3 (send talk-s :check-sym 3))
  (cond
   ((equal next-3 "@@@")
    (setf map-utm-e    (send talk-s :check-sym 5))
    (setf map-utm-n    (send talk-s :check-sym 5))
    (setf start-utm-e  (send talk-s :check-sym 5))
    (setf start-utm-n  (send talk-s :check-sym 5))
    (setf goal-utm-m-e (send talk-s :check-sym 5))
    (setf goal-utm-m-n (send talk-s :check-sym 5))
    (setf veh-id-str   (send talk-s :check-sym 10))
    (setf next-3       (send talk-s :check-sym 3))
    (cond
     ((equal next-3 "@@@")
      (setf map-utm-e    (convert-string-to-integer map-utm-e))
      (setf map-utm-n    (convert-string-to-integer map-utm-n))
      (setf start-utm-e  (convert-string-to-integer start-utm-e))
      (setf start-utm-n  (convert-string-to-integer start-utm-n))
      (setf goal-utm-m-e (convert-string-to-integer goal-utm-m-e))
      (setf goal-utm-m-n (convert-string-to-integer goal-utm-m-n))
      (setf veh-id       (convert-string-to-integer veh-id-str))
      (setf start-utm-e  (floor (/ (- start-utm-e map-utm-e)  100)))
      (setf start-utm-n  (floor (/ (- start-utm-n map-utm-n)  100)))
      (setf goal-utm-e   (floor (/ (- goal-utm-m-e map-utm-e) 100)))
      (setf goal-utm-n   (floor (/ (- goal-utm-m-n map-utm-n) 100)))
      (setf current-veh  (intern (string-append "VEH" veh-id-str)))
      (terpri)
      (princ "planning path")
      (terpri)
      (princ start-utm-e)
      (terpri)
      (princ start-utm-n)
      (terpri)
      (princ goal-utm-e)
      (terpri)
      (princ goal-utm-n)
      (setf *wave-paths* (cons
                          (add-seq-num
                           (add-id
                            (convert-to-utm
```

```
                    (append
                     (wave start-utm-e
                          start-utm-n
                          goal-utm-e
                          goal-utm-n
                          (symbol-value
                           (symbol-vale current-veh)
                           )
                          )
                     (list (list goal-utm-e goal-utm-n))
                     )
                    map-utm-e
                    map-utm-n
                    )
                   veh-id
                   )
                  )
                 *wave-paths*
                 )
            )
          )
         )
        )
       )
      )
     )
    (cond
     ((not (atom *wave-paths*))
      (setf *wave-paths* (send-waypoints *wave-paths*))
     )
    )
   )
  )

(defun new-utm (num map-org)
 (+ map-org (* num 100) (random 100))
 )
```

```
;;; -*- Package: USER; Mode: LISP; Syntax: Common-lisp -*-
;Title: lr-wave.lisp
;Author: Shannon
;Date: 20 May 1989
;Discription: This program is the implimentation of a wavefront search algorithm.
; (wave number-of-explored-cells touched-flag)
(defvar *cost-array*)
(defvar *center-cell*)
(defvar *s-wave*)
(defvar *g-wave*)
(defvar *array-size*)
(defvar *map-loaded*)
(defvar *map-array*)
(defvar *start-loc*)
(defvar *goal-loc*)
(defvar *parent-array*)


;(setf *map-size*)
(setf *start-loc* '(2 2))
(setf *goal-loc* '(10 10))

(defun parent-p(x y)
  (aref *parent-array* x y))

(defun set-new-cost(x y cost)
  (setf (aref *cost-array* x y) cost))

(defun set-new-parent(x y parent-x parent-y)
  (setf (aref *parent-array* x y) (list parent-x parent-y)))

(defun retrieve-cost(x y)
  (aref *cost-array* x y))

(defun retrieve-parent(x y)
  (aref *parent-array* x y))

(defun get-cost-from-map(x y)
  (aref *map-array* x y))

(defun load-map (mapsize map-e map-n mapfile veh-map)
  (setq input-stream (open mapfile :direction :input :byte-size 8 :characters nil))
  (setf map-loc (+ (* (floor (/ (- map-e 41000) 1000)) 10)
                   (* (floor (/ (- map-n 60000) 1000)) 3500)))
  (setf *array-size* (+ mapsize 2))
  (setf *map-array* veh-map)
  (do ((ycoord 0 (+ ycoord 1)))
```

```
    ((= ycoord *array-size*))
   (setf (aref *map-array* 0 ycoord) -2)
   (setf (aref *map-array* (- *array-size* 1) ycoord) -2))
  (do ((xcoord 0 (+ xcoord 1)))
    ((= xcoord *array-size*))
   (setf (aref *map-array* xcoord 0) -2)
   (setf (aref *map-array* xcoord (- *array-size* 1)) -2)
   )
  (do ((ycoord 1 (+ ycoord 1)))
     ((= ycoord (- *array-size* 1)))
   (file-position input-stream map-loc)
   (do ((xcoord 1 (+ xcoord 1)))
  ((= xcoord (- *array-size* 1)))
     (setf slope (read-byte input-stream))
     (setf slope (+ (/ (+ 0.0 slope) 2) 1))
     (cond
         ((> slope 15) (setf slope -2)))
     (setf (aref *map-array* xcoord ycoord) slope)
     )
   (setf map-loc (+ map-loc 350))
   )
  (close input-stream)
  (setf *map-loaded* 'yes)
  "READY"
  )

(defun wave(start-e start-n goal-e goal-n veh-map)
  (cond ((equal *map-loaded* 'yes)
 (setf *start-loc* (list start-e start-n))
 (setf *goal-loc* (list goal-e goal-n))
 (setf *map-array* veh-map)
 (read-terrain-data)
 (initial-expand)
 (normal-expand)
 (report-solution)
 )
(t
 'no-map-available)))

(defun report-solution()
  (append (reverse (follow-link (first *center-cell*) (second *center-cell*)))
  (cdr (follow-link (first *center-cell*) (third *center-cell*)))
  )
  )

(defun follow-link (pos1 pos2)
  (cond ((equal pos1 pos2)
```

```
 (list pos2))
(t
 (cons pos1
     (follow-link
 pos2
 (retrieve-parent (first pos2)
 (second pos2)))))))



(defun read-terrain-data()
  (let ((cost) (start-x) (start-y)
(goal-x) (goal-y))
    (setf *parent-array* (make-array (list *array-size* *array-size*)))
    (setf *cost-array* (make-array (list *array-size* *array-size*)))
    (copy-array-contents *map-array* *cost-array*)
    (setf start-x (first *start-loc*))
    (setf start-y (second *start-loc*))
    (setf goal-x (first *goal-loc*))
    (setf goal-y (second *goal-loc*))
    (set-new-parent start-x start-y start-x start-y)
    (set-new-parent goal-x goal-y goal-x goal-y)
    (set-new-cost start-x start-y -1) ; wave-name
    (set-new-cost goal-x goal-y 0) ; wave-name
    (pnnt 'done-terrain-classification)
    ))



(defun initial-expand()
  (do ()
      ((setf *s-wave* (init-expand -1 (list *start-loc*))))
    )
  (do ()
      ((setf *g-wave* (init-expand 0 (list *goal-loc*))))
    )
  )

(defun init-expand(wave-name wave)
; retrun: a-wave
  (first (expand-8 (car wave) wave-name)))


(defun expand-8 (pos wave-name)
  (let ((x (first pos))
                                              (y (second pos)))
    (orthog-expand (- x 1) y x y  wave-name
```

```
    (orthog-expand (+ x 1) y x y  wave-name
(orthog-expand x (+ y 1) x y  wave-name
 (orthog-expand x (- y 1) x y  wave-name
  (diag-expand (- x 1) (+ y 1) x y  wave-name
    (diag-expand (+ x 1) (+ y 1) x y  wave-name
(diag-expand (+ x 1) (- y 1) x y  wave-name
 (diag-expand (- x 1) (- y 1) x y  wave-name
    (list nil 0 0)))))))))))

(defun normal-expand()
 (do ()
    ((or (expand-s-wave)
  (expand-g-wave))
     (print 'wave-found))
   )
 )

(defun expand-s-wave()
 (set-new-s-wave (cycle-thru-wave *s-wave* -1 nil)))

(defun expand-g-wave()
 (set-new-g-wave (cycle-thru-wave *g-wave* 0 nil)))

(defun set-new-s-wave(new-wave-data)
 (setf *s-wave* (car new-wave-data))
 (>= (second new-wave-data) 1))


(defun set-new-g-wave(new-wave-data)
 (setf *g-wave* (car new-wave-data))
 (>= (second new-wave-data) 1))

(defun cycle-thru-wave(wave wave-name t-wave)
 (cond ((null wave)
(list t-wave 0 nil))
(t (let*
    ((pos (car wave))
     (x (first pos))
     (y (second pos))
     (a-parent (retrieve-parent x y))
     (dx (- x (first a-parent)))
     (dy (- y (second a-parent)))
     (wave-data (sub-expand dx dy x y wave-name t-wave))
     (wave-data1
(cycle-thru-wave (cdr wave) wave-name
 (add-back-to-wave pos wave-data))))
    (list (first wave-data1)
```

```
  (+ (second wave-data1) (second wave-data))
  nil)))))

(defun add-back-to-wave (pos wave-data)
  (if (>= (third wave-data) 3)
      (first wave-data)
      (cons pos (first wave-data))))


(defun sub-expand(dx dy x y wave-name wave)
  (cond ((equal dx 0)
 (sub-expand1 (+ y dy) x y wave-name wave))
((equal dy 0)
 (sub-expand2 (+ x dx) x y wave-name wave))
(t
 (sub-expand3 (+ x dx) (+ y dy) x y wave-name wave))))

(defun sub-expand1(ny x y wave-name wave)
  (diag-expand (+ x 1) ny x y wave-name
    (orthog-expand x ny x y wave-name
      (diag-expand (- x 1) ny x y wave-name (list wave 0 0)))))

(defun sub-expand2(nx x y wave-name wave)
  (diag-expand nx (- y 1) x y wave-name
    (orthog-expand nx y x y wave-name
      (diag-expand nx (+ y 1) x y wave-name (list wave 0 0)))))

(defun sub-expand3(nx ny x y wave-name wave)
  (orthog-expand nx y x y wave-name
     (diag-expand nx ny x y wave-name
(orthog-expand x ny x y wave-name (list wave 0 0)))))


(defun orthog-expand (x y px py wave-name wave-data)
  (a-expand x y px py 1.4142 wave-name wave-data))

(defun diag-expand (x y px py wave-name wave-data)
  (a-expand x y px py 1 wave-name wave-data))

(defun a-expand (x y px py amount wave-name wave-data)
  (if (not (parent-p x y))
      (set-new-parent x y px py))
  (let ((cost (retrieve-cost x y)))
    (cond
      ((and (equal cost -1)
                                                (equal cost (other-wave-p wave-name)))
       (setf *center-cell*
```

```
    (list (list x y)
   (retrieve-parent x y)
   (list px py)))
      (list (first wave-data)
    (+ (second wave-data) 1)
    (+ (third wave-data) 1)))
     ((and (equal cost 0)
   (equal cost (other-wave-p wave-name)))
      (setf *center-cell*
    (list (list x y)
   (list px py)
   (retrieve-parent x y)))
      (list (first wave-data)
    (+ (second wave-data) 1)
    (+ (third wave-data) 1)))
     ((and (equal (retrieve-parent x y) (list px py)) (> cost 0))
      (a-expand1 x y px py (- cost amount) wave-name wave-data))
     (t
      (list (first wave-data)
    (second wave-data)
    (+ (third wave-data) 1))))))



(defun a-expand1(x y px py new-cost wave-name wave-data)
  (cond ((> new-cost 0)
 (set-new-cost x y new-cost)
 wave-data)
(t
 (my-overflow x y px py new-cost wave-name
     (a-expand2 x y wave-name wave-data)))))

(defun a-expand2(x y wave-name wave-data)
  (set-new-cost x y wave-name)
  (list (cons (list x y) (first wave-data))
(second wave-data)
(+ (third wave-data) 1)))



(defun other-wave-p (wave-name)
  (if (equal wave-name 0)
     -1
     0))


(defun my-overflow (x y px py cost wave-name wave-data)
```

```
  (cond ((< cost 0)
 (let* ((nx (+ x (- x px)))
(ny (+ y (- y py)))
(cost1 (retrieve-cost nx ny))
(new-cost (+ cost cost1)))
   (if (not (parent-p nx ny))
      (set-new-parent nx ny x y))
   (cond ((and (equal (retrieve-parent nx ny) (list x y)) (> cost1 0))
  (cond ((> new-cost 0)
 (set-new-cost nx ny new-cost)
wave-data)
(t
 (set-new-cost nx ny wave-name)
 (my-overflow
   nx ny x y new-cost wave-name
   (list (cons (list nx ny) (first wave-data))
 (second wave-data)
 (third wave-data))))))
 (t
  wave-data))
   ))
(t
 wave-data)))
```

APPENDIX C USER INTERFACE

The user interface of any application program must be designed so that novice and experienced users alike can effectively operate the program with little or no help from user's manuals or other users. This is achieved by a thorough and efficient design of command line options, popup menus, dials, and the use of the mouse. This appendix provides instructions on starting up and running APS, both the vehicle simulator and the path planner, and navigating through the menus and operating controls of the system.

I. VEHICLE SIMULATOR[1]

The section covers the user interface to the vehicle simulator by describing starting procedures, the menu system, and platform controls.

A. COMMAND LINE OPTIONS[2]

The vehicle simulator is started by typing "aps" followed by any command line options and pressing RETURN. There are currently three options available from the command line.

- Network mode
- Test mode
- Silent mode

Selection of the network mode activates the networking capabilities of the program. In this mode update messages are sent and received from any other vehicle simulators as well as the path planner. Vehicle simulators operating on different machines will be able to share information regarding the other platforms. When a

---

[1]The main modifications to the MPS user interface are in the driving controls, weapon system controls and additional menu options. The entire user interface is documented here for completeness. Where the MPS interface is unmodified, it is an extract of Appendix A of [FICHTN88].

[2]The code that processes the command line arguments is contained in the file decode_arguments.c.

platform fires, changes guidance mode, or changes course, speed or altitude (FOGM only), a message is sent to all other vehicle simulators and to the AI agent updating the local database for the appropriate platform.

Selection of test mode bypasses some of the cosmetic portions of the program. Currently, the only part that is bypassed is the opening billboard sequence.

Selection of silent mode turns off the bell that rings to indicate acceptance of input from the user. This option is useful for demonstrations when the ringing would interfere with a verbal explanation of the program.

B. POPUP MENU SYSTEM[3]

Popup menus are the primary source of user control over the state of the program. There are currently 24 different popup menus that are used in various parts of the simulation. If a selection in a menu is not allowed or meaningful when the menu is displayed, the selection is displayed in lower case. Otherwise the selection is completely uppercase. Invalid selections are retained in the menu so that the menus always appear in the same order and format every time. If disallowed selections were omitted completely, users would tend to be overwhelmed by the number of different menu formats.

A menu is displayed and the selection always made by depressing the right mouse button. Roll-off menus are expanded by moving the cursor arrow to the right when a menu item with a roll-off submenu (such selections have a small arrow on the right-hand side) is highlighted. The following is a detailed explanation of each menu.

---

[3]The code for defining all static popup menus is contained in the file makepopups.c. Code for displaying and processing menu selections is contained in the following files: do_main.c, do_driving_menu.c, do_flying_menu.c, do_change_speed.c, do_intros.c, do_pathops.c, do_quitting.c, do_select_area.c, do_the_add.c, do_the_defaults.c, do_the_delete.c, do_the_select.c, and select_sight.c.

1. Opening Menus

There are two menus that make up the opening menu set. These menus are called OPENING_ONE and OPENING_TWO. Each of these menus contain the same four selections as follows:

- DISPLAY INSTRUCTIONS
- GO TO SELECT AREA MENU
- EXIT THE PROGRAM
- ENTER 4SIGHT (RESIZE OPTIONS)

OPENING_ONE allows the user to select any one of these options but OPENING_TWO disallows the first option. OPENING_TWO is displayed if the user is currently looking at the instruction page.

The first selection displays a page of instructions onthe user interface. If the instruction page is being displayed or the user wishes to bypass the instruction page, the GO TO SELECT AREA MENU selection will do just that. To exit the program, the user must select EXIT THE PROGRAM and a small menu will be displayed with the following selections:

- RETURN TO WHERE YOU WERE
- REALLY QUIT

If the user desires to resize or move the simulation's windows, the option ENTER 4SIGHT (RESIZE OPTIONS) will allow him to accomplish it. After selecting the option, the windows will be cleared to white and the user can click on the menu bar and move or resize as desired using normal window manager functions.

2. Select Area Menu

The select area menu is active whenever the 35 KM 2D map is displayed. It contains the following options:

- SELECT AN AREA OF THE MAP
- GO TO MAIN MENU
- EXIT THE PROGRAM

- ENTER 4SIGHT (RESIZE OPTIONS)
- COLOR SCHEME - BROWN RAMP
- COLOR SCHEME - MULTIPLE COLORS
- COLOR SCHEME - GREY RAMP
- COLOR SCHEME - RED RAMP
- COLOR SCHEME - GREEN RAMP
- COLOR SCHEME - BLUE RAMP
- GO TO INTRODUCTION SCREEN

Selecting GO TO MAIN MENU will take the user to the main menu which is the next logical place to go after selecting a 10KM area in which to operate.

The color scheme selections change the way the terrain is colored. Each color scheme has eight different colors that are based on the elevation at that location. The simulation actually uses 16 colors to create a checkerboarding effect, however the user is only shown the eight primary colors in the color ramp.

The last selection allows a user to return to the introduction screens if he desires.

3. Main Menu

The main menu contains the following ten selections:

- PLACE DEFAULT SET OF PLATFORMS
- ADD A PLATFORM
- DELETE A PLATFORM
- SAVE PLATFORMS TO A FILE
- SELECT A PLATFORM TO OPERATE
- ENTER 4SIGHT (RESIZE OPTIONS)
- SELECT ANOTHER AREA OF THE MAP
- PERFORM PATH OPERATIONS
- OBSTACLES ON/OFF ⇒
- EXIT THE PROGRAM

Selecting the first option (PLACE DEFAULT SET OF PLATFORMS) will display another menu called DEFAULT_MENU. This menu contains 6 selections as follows:

- ENTER THE FILENAME FOR YOUR PLATFORMS
- CONVOY - 10 GROUND PLATFORMS
- CONVOY - 10 GROUND & 1 FOGM PLATFORM
- JEEPS - 20 IN A ROW
- DR. ZYDA'S CONVOY
- DR. ZYDA'S WILDMAN DEFAULTS

If the user selects the first option, a small window is displayed on the screen which prompts the user for the filename. If valid information is found in the file, the appropriate platforms are added to the simulation. The main menu is then redisplayed.

Selection of any other option on the DEFAULT_MENU results in the addition of predesignated platforms in predesignated locations. These selections are useful for demonstration purposes and for persons interested in getting some platforms on the screen very quickly.

The information for the default sets of platforms is contained in data files that are read when indicated by a menu selection. The complete path for these files is contained in the header file "files.h".

The next option on the main menu is ADD A PLATFORM. Selecting this option displays the following menu:

- ADD A COVERED JEEP
- ADD AN OPEN JEEP
- ADD A TRUCK
- ADD A TANK
- ADD A TOW VEHICLE
- ADD A FOGM MISSILE
- ADD AN ATTACK HELICOPTER
- ADD AN OBSTACLE

If a moving platform is selected (jeep, truck, tank, TOW, attack helicopter, or FOGM), menus are displayed requesting an initial speed and direction for the platform. If an obstacle is requested, then the speed and direction menus are bypassed. The FOGM missile defaults to an initial altitude of 50 meters above the terrain at the point where it is placed. After completing the selections, an icon is placed in the center of the screen that resembles the selected platform or obstacle. The user can then move the icon with the mouse and place the platform by clicking the right mouse button. After placing the icon on the screen, the main menu is displayed once again.

Selecting the DELETE A PLATFORM option displays the following menu:

- DELETE A SINGLE PLATFORM
- DELETE ALL PLATFORMS ON THE SCREEN

If the user wants to delete one platform, an X cursor is displayed and the user can click on the desired platform. If the user wants to delete all the platforms on the screen, the following menu is displayed:

- NO, DO NOT DELETE ALL THE PLATFORMS
- YES, DELETE ALL PLATFORMS

The appropriate selection from this menu either cancels the operation or executes it. This menu prevents a user from deleting vehicles that he may not really want to delete.

If the user has placed platforms on the screen and wishes to save them to a file, then the main menu selection SAVE PLATFORMS TO A FILE accomplishes this. A window opens that prompts the user for the filename. If the path is correct, the platforms are saved to the file.

The next selection from the main menu is SELECT A PLATFORM TO OPERATE. If the user selects this option, the following menu is displayed:

- ZOOM IN TO ANY LEGAL GRID SQUARE
- SELECT A PLATFORM TO OPERATE RIGHT NOW

The zoom option is usually necessary if platforms are close to each other and the individual icons overlap. By zooming into the 1x1 kilometer grid square, the user can more easily select the platform he desires. If the platform the user wants to operate is clearly visible, then the second selection allows the user to select a platform immediately.

The SELECT ANOTHER AREA OF THE MAP option returns to the SELECT_AREA menu and redisplays the 35KM map.

Selecting PERFORM PATH OPERATIONS from the main menu displays a submenu containing up to four path manipulation functions. These functions are:

- DISPLAY PATHS ON/OFF ⇒
- CONSTRUCT PATH
- DELETE PATH
- ASSIGN VEHICLE TO PATH

The last two options are not displayed if there are no paths to delete or no vehicles to assign to a path. The first selection is a toggle that turns the display of paths on the 10KM map on and off. The other selections allow manipulation of paths. When a function is invoked by selecting it, specific instructions are displayed in the lower right menu window.

4. Operating Menus

Operating menus are available when a platform has been selected and is being driven by the user. They generally affect the characteristics of the 3D terrain display or how the vehicle is being controlled.

a. Driving Menu. This menu is called OPERATE_DRIVE. It contains ten selections:

- DO NOTHING
- RETURN TO MAIN MENU
- CHANGE ALL PLATFORMS' SPEEDS
- EXIT THE PROGRAM
- ENTER 4SIGHT (RESIZE OPTIONS)
- POP WINDOWS
- CHANGE VIEW
- ADVANCED OPTIONS ⇒
- AUTOPILOT ON/OFF ⇒
- GUIDANCE ON/OFF ⇒

The first selection is provided in case the user pushes the right mouse button and he does not desire to do anything. The second selection returns the user to the main menu.

The third selection causes another menu to pop up that allows the user to select a speed for all the platforms currently in the simulation. The allowable speeds are from zero to 65 miles per hour. There is also a selection that will do nothing and return directly to the simulation. Changing all the speeds is convenient when the user wants to have a convoy of platforms proceed at identical speeds. Also, by selecting zero miles per hour, all platforms are effectively frozen and their configuration can be studied by viewing them from a FOGM missile or other platform.

The POP WINDOWS selection brings the four windows of the simulation into view if any of them are obscured from view by other processes that are running on the machine.

If the CHANGE VIEW option is selected, a submenu containing different operating modes is presented. All platforms have at least three options:

- NORMAL VIEW - Normal commander's view, all dials including course and speed are active.

- DRIVER'S STATION - Activates mouse joystick (Figure C-1). for driving the platform. In this mode moving the mouse move the *steering cursor* which controls the steering and throttle. The corresponding course and speed dials are deactivated.
- BINOCULARS - Gives view through a pair of variable power binoculars.

An additional selection is presented for each weapon system type and munition combination carried by the platform, i.e., for a TOW vehicle a TOW selection is displayed along with the normal three views.

The ADVANCED OPTIONS selection brings up the following menu:

- TOGGLE SINGLE/DOUBLE BUFFER MODE
- TARGETING MODE TEST (ONCE)
- TERRAIN DRAWING OPTIONS

The first selection toggles the graphics hardware between singlebuffer and doublebuffer modes. In doublebuffer mode, all drawing is done in a separate area of memory from the display memory. When the function **swapbuffers()** is called, the pointer to this area and the pointer to the display buffer are switched, thereby swapping the new picture for the old picture. This is how smooth motion is simulated. If a user is interested in what order the individual picture elements are drawn on the screen, then by selecting singlebuffer mode, he can see the pictures while they are being drawn.

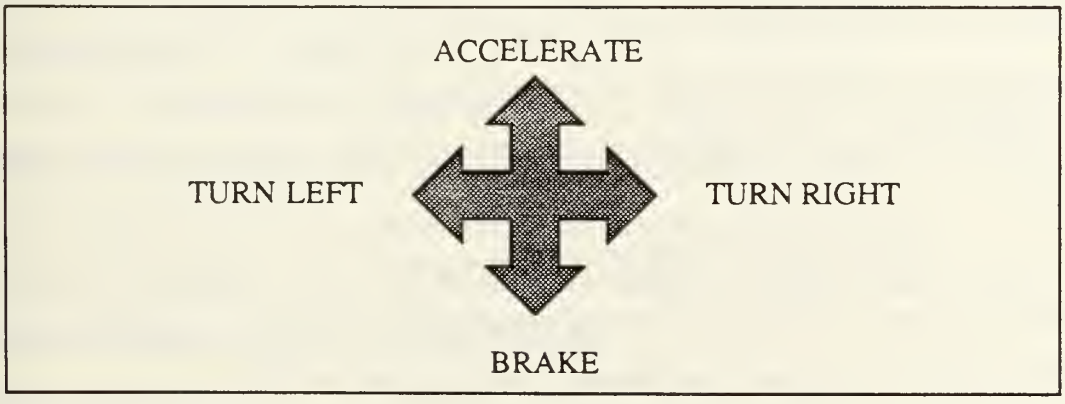

Figure C-1 Mouse Steering Cursor

Targeting mode test allows a user to see how the simulation determines if a target is in the crosshairs of the FOGM missile during targeting. After selecting the option, the next time targeting is attempted, the view will be cleared to white and all visible platforms will be drawn without lighting, shading, or hidden surface removal. The resulting picture is displayed for three seconds and then normal operation commences. This option is reset each time it is used.

The TERRAIN DRAWING OPTIONS option is a roll-off menu. When the user moves the cursor towards the right side of the words TERRAIN DRAWING OPTIONS, the following menu is displayed:

- DETAILED TERRAIN
- DISTANCE ATTENUATION - NORMAL
- DISTANCE ATTENUATION - BOUNDARIES DISPLAYED

The default terrain drawing option is DISTANCE ATTENUATION - NORMAL. This drawing option establishes three zones in front of the driven platform and reduces the number of polygons that are displayed in each zone. The zone closest to the viewer is displayed with 100x100 meter polygons, the greatest resolution available. The next zone uses 200x200 meter polygons and the last zone uses 400x400 meter polygons. The selection DISTANCE ATTENUATION - BOUNDARIES DISPLAYED draws the boundaries between zones in cyan so the user can see where they are. The selection for DETAILED TERRAIN draws 100x100 meter polygons throughout the three zones. Users notice a significant decrease in the frames per second rate when this option is selected. If singlebuffer mode is also enabled during detailed terrain drawing, the algorithm that is used to draw the terrain becomes more obvious.

The GUIDANCE ON/OFF toggles the guidance mode of the currently selected platform. It invokes the actions described in paragraph C of chapter three. A indicator light in the upper right window is also toggled to reflect the current guidance mode.

The AUTOPILOT ON/OFF option works much the same. It toggles the platform's autopilot and its indicator light on and off.

b. Flying. There are three menus that make up the flying menu set. These menus are called OPERATE_FLY_ONE, OPERATE_FLY_TWO, and OPERATE_FLY _THREE. This menu contains the seven selections as follows:

- DO NOTHING
- DETACH/RESUME OPERATING
- RETURN TO MAIN MENU
- CHANGE ALL PLATFORMS' SPEEDS
- EXIT THE PROGRAM
- ENTER 4SIGHT (RESIZE OPTIONS)
- TOGGLE TARGET TRACKING
- ADVANCED OPTIONS

Many of these options are exact duplicates of the options on the driving menu. However, the DETACH/RESUME OPERATING and TOGGLE TARGET TRACKING options are different.

The DETACH/RESUME OPERATING option allows a user to detach the cursor from the simulation while flying. During flying, the cursor is restricted to the simulation window because the mouse controls where the nose camera of the FOGM missile is pointed. Using this option, the user can point the camera where he wants to look and then free the mouse. To return to the simulation, the user must select the same option once again.

If the user has a ground platform in the crosshairs of the FOGM missile and he wants to target it, he must make the TOGGLE TARGET TRACKING selection from the menu. If a platform was in the crosshairs, then the missile will lock on and track the platform. If the user wants to release the missile from tracking mode then another selection will turn off target tracking.

C. DIALS[4]

The dial box that is supplied by SGI has eight dials numbered from zero to seven. They are organized in two columns and four rows. The numbering scheme is from left to right, bottom to top so the lower left dial is zero, the lower right is one and the upper right is seven.

The Autonomous Platform Simulator uses these dials in basically three configurations; one for driving a platform that has *no* weapon system, a second for driving a weapon equipped platform and a third for flying the FOGM. When the vehicle i being driven using the mouse joystick, the course and speed dials are inactive. When looking through the weapon sight of a platform dials one and three affect the azimuth and elevation respectfully of the weapon system. When in normal view mode dials six and seven perform this function and the weapon is controlled independently of vehicle course or viewing angle.

1. Driving Dial Configuration

The dials for driving (Figure C-2) are configured as follows:

- DIAL 0 - Course
- DIAL 1 - Viewing direction or weapon azimuth if a sight is active
- DIAL 2 - Speed
- DIAL 3 - Viewing elevation or weapon elevation if a sight is active
- DIAL 4 - Hour of the day
- DIAL 5 - Month of the year
- DIAL 6 - Traverse weapon system when not looking through sight
- DIAL 7 - Elevate weapon system when not looking through sight

The course is the direction of travel of the platform which is displayed in degrees. The viewing direction is the direction the driver's head is looking left to right in relation to the course. When the course is changed, the viewing angle changes accordingly. Speed is the speed of the platform in miles per hour. View elevation

[4]The code for initializing the dials is contained in the following files: setcontrols.c and setcontrols_fogm.c. Code for handling input from the dials' movements is contained in the following files: handlecontrols.c, handlecontrols_fogm.c, and handlecontrols_partial.c.

moves the driver's view up and down. The hour of the day and month of the year determine the location, color, and intensity of the sun. Figure C-2 is a picture of the dial box with the dials labeled for driving.

2. *Flying Dial Configuration*

The dials for flying are configured as follows:

- DIAL 0 - Course
- DIAL 1 - Altitude
- DIAL 2 - Speed
- DIAL 3 - Not Used
- DIAL 4 - Hour of the Day
- DIAL 5 - Month of the Year
- DIAL 6 - Not Used
- DIAL 7 - Not Used

Many of the dials are identical to the driving dial configuration except for altitude which is self-explanatory. Figure C-3 is a picture of the dial box with the dials labeled for flying.

D. Mouse[5]

The mouse has many uses throughout the simulation. Its use can be broken down into basically six groups:

- Popup menu activation and selection
- Operating area selection
- Platform icon placement and selection
- FOGM missile nose camera control
- Mouse joystick driving control
- Weapon rangefinder and firing controls

[5]Code for handling the operations of the selections is contained in the file select_area_menu.c. Code for handling platform icon placement is contained in the files do_the_add.c and addveh.c. Code for driving using the mouse as a joystick is contained in setup_for_driving.c. Code for handling FOGM missile nose camera control is contained in the files handlecontrols_fogm.c and handlecontrols_partial.c.

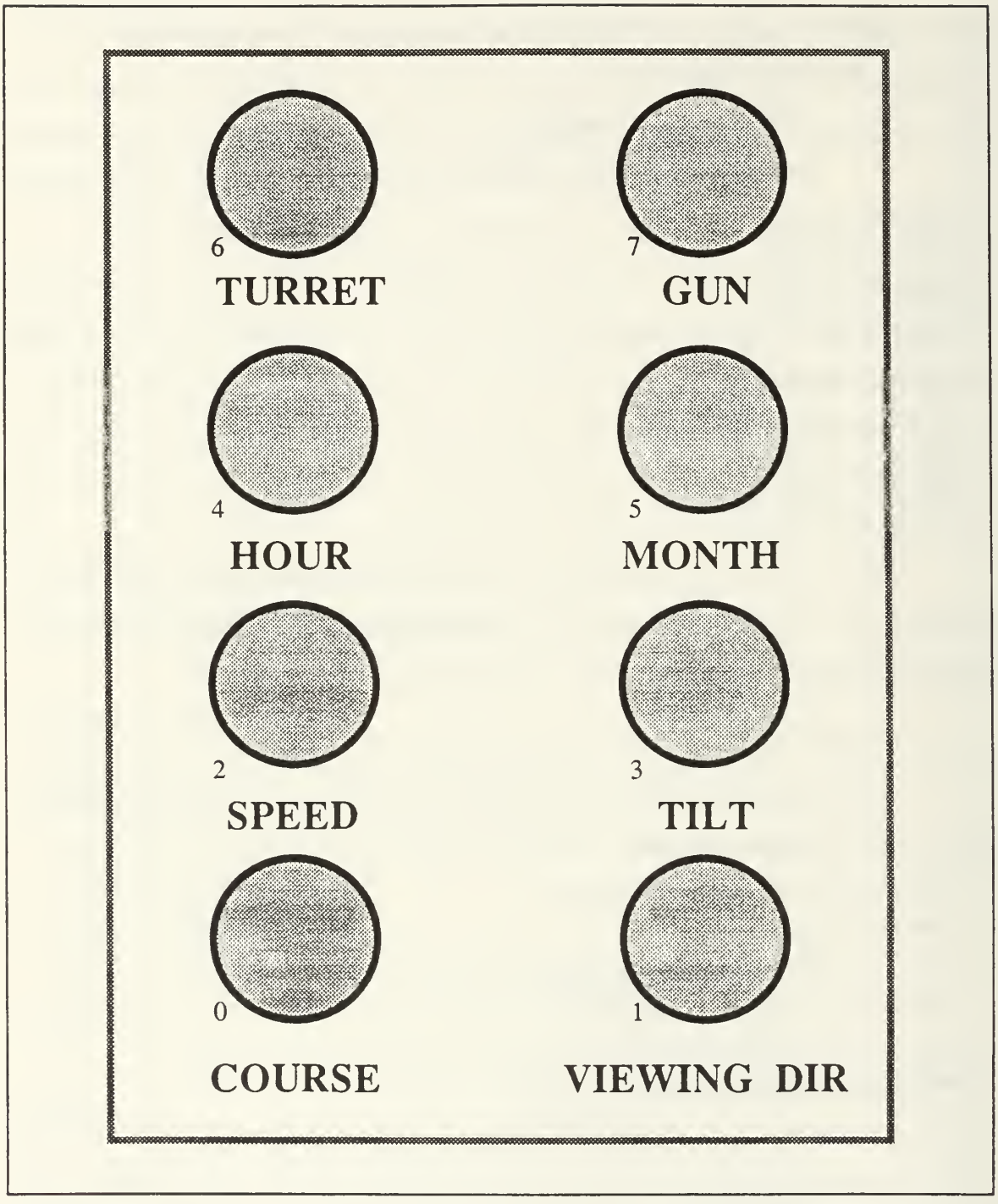

Figure C-2 Dial Box With Dials Labeled For Driving

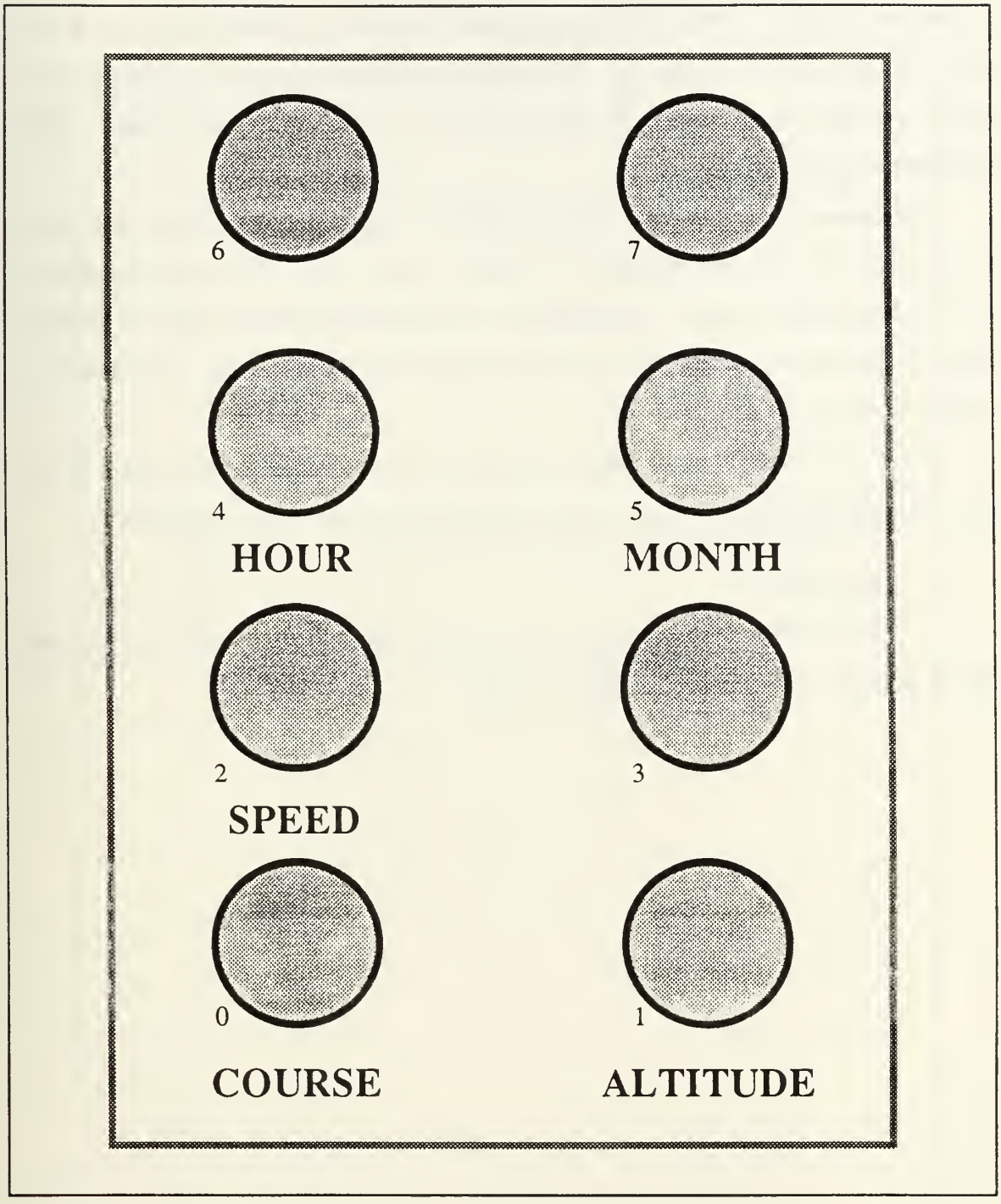

Figure C-3 Dial Box With Dials Labeled For Flying

When operating a platform using the dials or mouse joystick the left and middle mouse buttons control the magnification of the view by zooming OUT or IN respectively as shown in Figure C-4. When looking through the sight of a weapon system the left and middle mouse buttons function as a rangerfinder and trigger. This arrangement is shown in Figure C-5.

The mouse is used throughout the simulation to activate popup menus and to select options. One of these options is to select an area from the large database. A 10x10 kilometer red square is displayed on the 35x35 kilometer database and the mouse is used to move the square to the desired location. Platforms are placed and selected on the screen with the mouse.

The nose camera of the FOGM missile is controlled with the movement of the mouse. This gives the user very fine control over targeting and viewing direction.

E. Keyboard[6]

The keyboard is only used to accept filenames from the user. All other user input is through the popup menus, dials, or mouse.

[6]Code for handling filename input is contained in the files get_name.c and do_char.c.

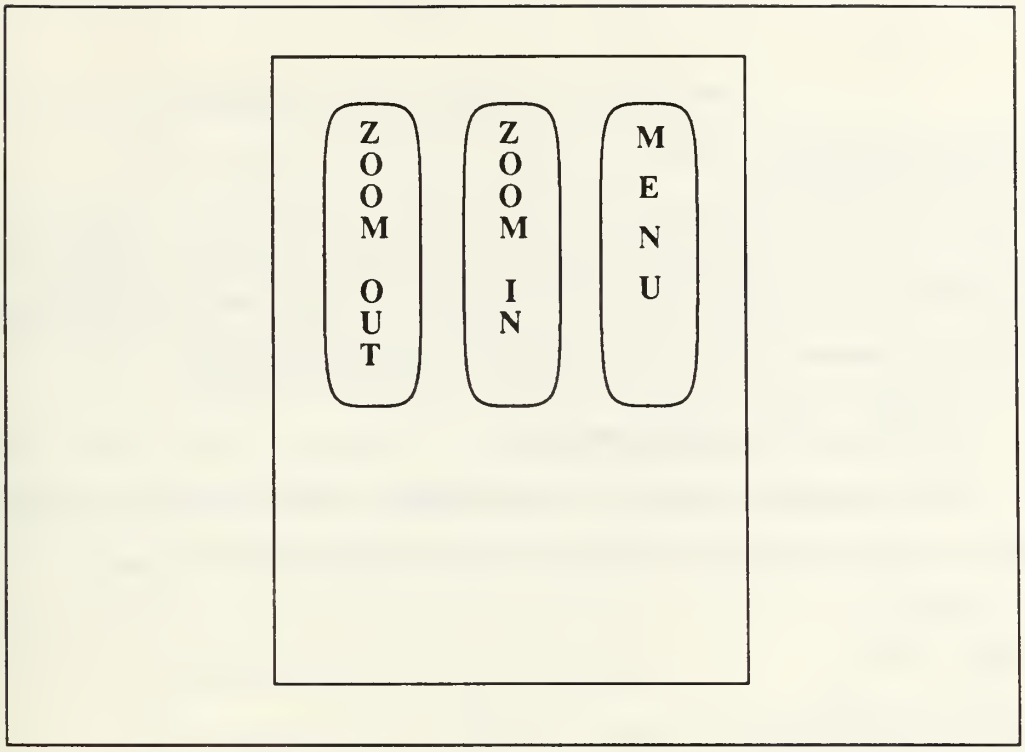

Figure C-4 Mouse Button Assignments - Normal View

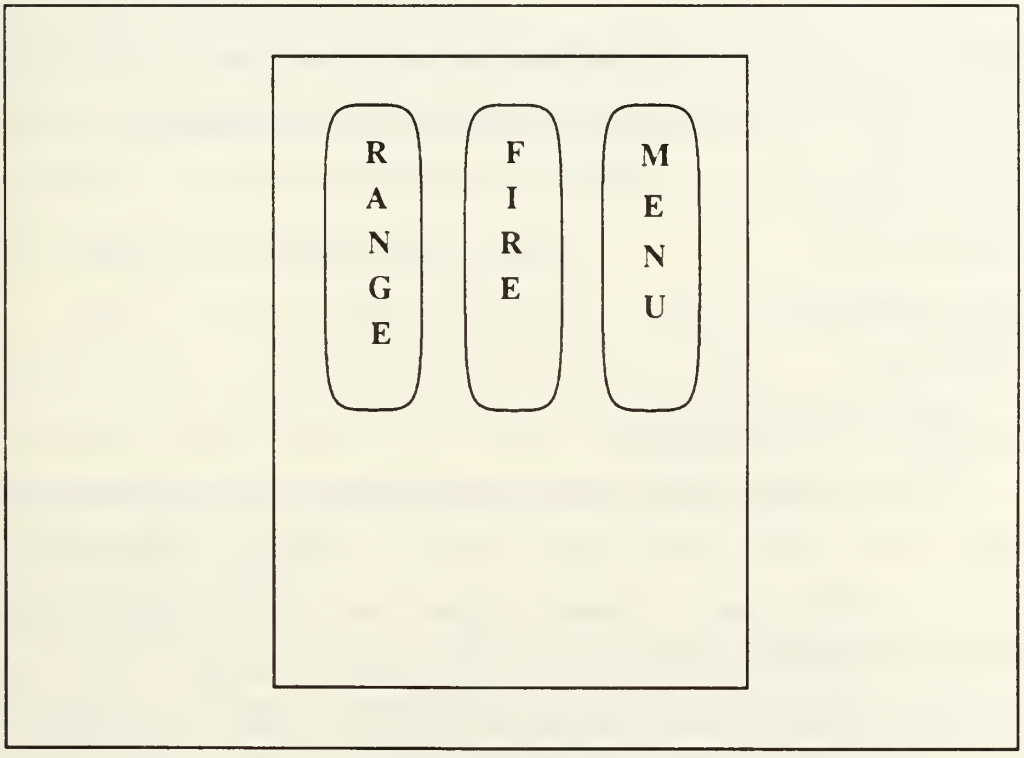

Figure C-5 Mouse Button Assignments - Weapon Sight

II. PATH PLANNER

The path planner portion of APS is not a stand alone process, it requires the vehicle simulator to be running. This section covers how to run the path planner portion of the vehicle simulator by describing starting and stopping procedures.

A. INITIAL REQUIREMENTS

The path planner requires that seven files be loaded across two Symbolics workstations. The following files are required on SYM4, where ART resides.

- pp-control.art
- irisflavor3.lisp
- chaosflavor.lisp
- comm-functions.lisp
- clock-functions.lisp
- def-interface.lisp

The above files do not have to be loaded to begin with, but must be available for file access. The following files are required on another Symbolics workstation.

- big-slope.bin
- search-control.lisp
- chaosflavor.lisp
- comm-functions.lisp
- lr-wave.lisp

B. START PROCEDURES

The path planner requires several preconditions to run properly. Since the path planner is not a stand alone program, APS must be brought up first in network mode. The path planner may be started anytime after APS has passed the initial screen display. Starting the path planner is divided into two sections. These sections are starting the path planner control program, and starting the search control program.

1. *Starting the Path Planner Control Program*

On SYM4, enter ART by typing the SELECT: button, then typing A. Load pp-control.art in the ART shell. Reset ART by clicking the left mouse button on reset. Left mouse click on run. The program will query the user as to which Symbolics workstation the search algorithm is loaded. After ensuring that the search control program is started on the other Symbolics workstation, select the appropriate letter. The program will then query the user as to which IRIS graphics workstation the vehicle simulation is running on. After ensuring that the simulation is already running in the network mode, select the appropriate letter from the menu. The path planner is now running on its own and needs no further user interaction.

2. *Starting the Search Control Program*

The search control program is loaded onto any Symbolics workstation, other than SYM4 where the path planner control program is loaded. To start the search program, load search-control.lisp. Then in the LISP listener enter (start-search-control). The program will respond by loading all of its communications and search files, then initiate a wait for communications from the path planner control program on SYM4. No further user interaction is required.

C. STOPPING THE PATH PLANNER

When the user is finished with the path planner, it is halted by using the META, CONTROL, and ABORT keys simultaneously. Next, on SYM4, the user enters the dynamic LISP listener and sets the user package to ACU (ART Common User), and clears the communications paths by entering the following:

- (scl:send talk-i :stop-iris)
- (scl:send talk-s :stop)

The search control program is halted in an identical manner as the path planner control program, but there is no need to enter a special LISP package to clear the communications path. The communications path for the search control program is cleared by entering **(send talk-s :stop)**.

APPENDIX D KNOWN BUGS and SUGGESTED CODE IMPROVEMENTS

1. The timer is currently reset when a vehicle is selected/reselected to operate from the main menu. This causes errors in timed events on the event list such as rounds in flight, safety reset, etc..

2. The Cobra attack helicopter is controlled the same as a ground vehicle.

3. Guidance for LOS guided weapons, specifically the TOW, uses the current weapon azimuth and elevation, not the parameters at the moment of firing like a ballistic round. This is correct but still fails to move the round onto the LOS at the crosshairs of the sight reticle. In **check_round_in_flight()**, the round should be moved to its new updated position by moving towards the *current* point of aim while being kept within the turning limits of the missile control system.

4. A separate eye position should be added to provide additional viewpoints on each vehicle. Each vehicle should have a eye position for: normal view (TC), driver's view, and weapon view. This should be accomplished by adding the following data structure:

```
#define  TC_POSITION          0
#define  DRIVER_POSITION      1
#define  WEAPON_POSITION      2
         .
         .
float eye_position[vehtype][view_position][x,y,z]
```

5. The FOGM controls no longer work correctly. The pan direction is reversed and the course is fixed at 90 degrees.

6. The **network(SEND_END_MESSAGE)** function causes remote simulators to crash. Since net ids are now unique, the range of ids can no longer be calculated. Therefore, the terminating simulator must send a delete message for each of its local platforms when terminating.

7. In some cases, the autopilot will cause a platform to endlessly orbit the goal. Normally a platform approaches a guide point head-on and stops or turns. If

the vehicle is outside the stopping distance and facing away from the goal when the autopilot is engaged, then the vehicle can get into a situation where it passes by the guide point before it completes its turn to head for it. This results in a circular path around the guide point. This should be taken care of when the autopilot is made more accurate to handle obstacle avoidance.

8. The display limiting algorithm in drawterrain doesn't work properly for the extremely narrow field-of-view used for the 13X TOW sight. At certain angles not enough terrain is drawn so some blue sky background shows through.

9. Calculating surface normals for 100 squares across and up requires 101 elevation data points. The 101st elevation doesn't exist, resulting in bad normals along the top row and right column. This was fixed temporarily by extending the 100th elevation out to also be the 101st elevation which creates a light band of terrain in these border areas. The algorithm should be changed to either get the 101st elevation or extrapolate it based on 99th and 100th elevations.

10. All matching of platform ids is done by linear search through the platform list. This is not a problem with only a handful of vehicles, but would cause serious delays for a more realistic number of platforms. This should be replaced by a hash table using platform id.

LIST OF REFERENCES

[BARROW&88] Barrow, Theodore H., Yurchak, John M., Zyda, Michael. J., *Distributed Computer Communications in Support of Real-Time Visual Simulations*, Technical Report, Naval Postgraduate School, Monterey, California, September 1988

[BIHARI&89] Bihari, Thomas E., Walliser, Thomas M., and Patterson, Mark R., "Controlling the Adaptive Suspension Vehicle", *IEEE Computer*, pp 59-65, June 1989.

[DODSCI83] U. S. Department of Defense Advanced Research Projects Agency, *Strategic Computing: New-Generation Computing Technology: A Strategic Plan for Its Develpement and Applications to Critical Problems in Defense*, DARPA Washington DC, 23 October 1983

[FELHOE89] Felhoelter, Dennis G., *A Graphics Facility for Integration, Editing, and Display of Slope, Curvature, and Contours From a Digital Terrain Elevation Database,* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1988.

[FEYNMN63] Feynman, Richard P., Leighton, Robert B., and Sand, Michael, *Lectures on Physiscs,* v. 1, California Institute of Technology, 1963.

[FICHTN&88] Fichten, Mark A., Jennings, David H., *Meaningful Real-Time Graphics Workstation Performance Measurements,* M.S. Thesis, Naval Postgraduate School, Monterey, California, November 1988.

[FRANK&69] Frank, A. A., and McGhee, Robert B., "Some Considerations Relating to the Design of Autopilots for Legged Vehicles", *Journal of Terramechanics*, v.6, n.1, pp 23-35, 1969.

[FU&87] Fu, K.S., Gonzales, R. C., Lee, C. S. G., *Robotics: Control, Sensing, Vision, and Intelligence,* McGraw-Hill Book Company, 1987.

[GOODPA87] Goodpasture, Richard P., *A Computer Simulation Study of an Expert System For Walking Machine Motion Planning*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987.

[HEARN&86] Hearn, Donald and Baker, M. Pauline, *Computer Graphics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[INFRNC85] Inference Corporation, *ART Reference Manual*, Los Angeles, California, 1985.

[INTEL86] IntelliCorp, *KEE Software Development System User's Manual*, version 3.0, Mountain View, California, 1986.

[JANES87] Jane's Publishing Company Ltd., *Armour and Artillery 86-87*, London, England, 1987.

[KUAN84] Kuan, D. T., "Terrain Map Knowledge Representation for Spatial Planning", *Proceedings of the IEEE Computer Security Conference on Autonomous Vehicle Applications*, 1984.

[KWAK&88] Kwak, Sehung. and McGhee, Robert. B., *Rule-based Motion Coordination for the Adaptive Suspension Vehicle*, Technical Report, Naval Postgraduate School, Monterey, California, May 1988.

[LOWRIE86] Lowrie, J., *The Autonomous Land Vehicle Program*, Martin Marietta, Denver Aerospace, Denver, Colorado, December 1985.

[MARION70] Marion, Jerry B., *Classical Dynamics of Particles and Systems*, Academic Press, College Park, Maryland, 1970.

[METEA&87] Metea, M. B., and Tsai, J., "Route Planning for Intelligent Autonomous Land Vehicles Using Hierarchical Terrain Representation", *Proceedings of the IEEE Conference on Robotics and Automation*, 1987.

[MCNKLE&88] McConkle, Corinne and Nelson, Andrew H., *A Prototype Simulation System for Combat Vehicle Coordination and Motion Visualization*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1988.

[MUNSON89] Munson, Steven, *Integrated Support for Manipulation and Display of 3D Objects for the Command and Control Workstation of the Future*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[NITAO&88] Nitao, John J. and Parodi, Alexander M., "A Real-Time Reflexive Pilot for an Autonomous Land Vehicle", *IEEE Control Systems Magazine*, pp 14-23, Febuary 86.

[NIZOLK&89] Nizolak, Joseph, P. Jr., and Drummon, William T., *A Graphics Workstation Field Artillery Forward Observer Simulator Trainer*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[OLIVER&87] Oliver, Michael R., and Stahl, David J., *Interactive, Networked, Moving Platform Simulators*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987.

[RESNCK&67] Resnick, Robert, and Halliday, David, *Physics, Part I*, John Wiley & Sons, Inc., 1967.

[RICHBG&87] Richbourg, R. E., Rowe, N. C., Zyda, M. J., and McGhee, R. B., "Solving Global, Two-Dimensional Routing Problems Using Snell's Law and A* Search", *Proceedings of the IEEE International Conference on Robotics and Automation*, April 1987.

[RICHBG87] Richbourg, R. F., *Solving a Class of Spatial Reasoning Problems: Minimal-Cost Path Planning in the Cartesian Plane*, Doctoral Thesis, Naval Postgraduate School, Monterey, California, June 1987.

[ROSS89] Ross, R. S., *Planning Mininum-Energy Paths in an Off-Road Environment With Anisotropic Traversal Costs and Motion Constraints*, Doctoral Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[ROWE87] Rowe, Neil C., *Roads, Rivers, and Rocks: Optimal Two-Dimensional Route Planning Around Linear Features for a Mobile Agent*, Technical Report, Naval Postgraduate School, Monterey, California, June 1987.

[ROWE&88] Rowe, Neil C. and Ross, Ron S., *Optimal Grid-free Path Planning Across Arbitrarily-Contoured Terrain With Anisotropic Friction and Gravity Effects*, Technical Report, Naval Postgraduate School, Monterey, California, November 1988.

[ROWE88B] Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice-Hall, Inc, 1988.

[SGIIUG87] Silicon Graphics Inc., *IRIS User's Guide*, v. 1, Mountain View, California, 1987.

[SGI4UG88] Silicon Graphics Inc., *4Sight User's Guide*, v. 1, Mountain View, California, 1988.

[SMITHD&87] Smith, Douglas B., and Streyle, Dale G., *An Inexpensive Real-Time Interactive Three-Dimensional Flight Simulation System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

[SYMBOL88] Symbolics, Inc., Symbolics Reference Manuals, Cambridge Massachusetts, 1988.

[TAN86] Tan, Chaim Huat, *A Simulation Study of An Autonomous Steering System for On-Road Operation of Autonomous Vehicles*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1986.

[WEISBN&89] Weisbin, C. R., and others, "Autonomous Mobile Robot Navigation and Learning", *IEEE Computer*, pp 29-35, June 1989.

[WINN&89] Winn, Michael C., and Strong, Randolph P., *The Moving Platform Simulator II: A Networked Real-Time Simulator With Intervisibility Displays,* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1989.

[ZYDA&88] Zyda, Michael J., and others, "Flight Simulators for Under $100,000", *IEEE Computer Graphics and Applications*, pp19-27, January 1988.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Robert B. McGhee Naval Postgraduate School Code 52, Department of Computer Science Monterey, CA 93943-5100	2
3.	Dr. Michael J. Zyda Naval Postgraduate School Code 52Mz, Department of Computer Science Monterey, CA 93943-5100	2
4.	Maj. William A. Teter 2 Mervine Street Monterey, CA 93940	2
5.	Cpt. Larry R. Shannon Star Route Entiate, WA 98801	2
6.	Dr. Byron Dean United States Army Intelligence Center and School Attention: Scientific Advisor Fort Huachuca, AZ 85613	1
7.	U.S. Army AI Center HQDA, DCSA, DSMA ATTN:CSDS-AI(MAJ TETER) Pentagon, RM 1D659 Wash, DC 20310-0200	2

8.	Commandant of the Marine Corps Code TE 06 Headquarters, United States Marine Corps Washington, D.C. 20380-0001	1
9.	Mr. Mike Tedeschi United States Army Combat Developments Experimentation Center Attention: ATEC-D Fort Ord, CA 93941	2
10.	Information Technology, Code 037 Naval Postgraduate School Monterey, CA 93943-5100	2

Thesis
S433424 Shannon
c.1 An Autonomous Platform Simulator (APS).